University of California
Santa Cruz

Optimal schedules for tasks with stochastic runtimes

A project submitted in partial satisfaction
of the requirements for the degree of

Master of Science

in

Statistics and Applied Mathematics

by

Niranjan Vissa

March 2016

The Project of Niranjan Vissa

is approved:

Professor David Draper

Professor Herbert Lee

ACKNOWLEDGEMENTS

I would like to acknowledge the guidance, wisdom and patience of Prof. David Draper without whom this project could not have been completed. I would also like to thank Prof. Herbert Lee for taking the time to read initial versions of this report and provide many valuable comments.

Contents

List of Tables

List of Figures

Notation

$\mathcal{I}$	Set of instances available to process tasks. Each instance is associated with a *type* Y and a *cost* C.
$\mathcal{T}$	Set of input tasks that must be processed. Each task is associated with a *length* L.
B	Benefit (specified in dollars) paid out when all tasks in $\mathcal{T}$ are completed by the deadline. If $\mathcal{T}$ is not complete by the deadline, $B = \$0$.
C_j	Cost per hour of instance I_j (specified in dollars/hour)
D	Deadline (specified in hours) by which all tasks in $\mathcal{T}$ must complete processing
L_i	Length of task T_i
M_S	Makespan of schedule S to process the set of tasks $\mathcal{T}$
R_j	Total runtime of all tasks assigned to instance I_j
R_{ij}	Runtime of task T_i on instance I_j; known only when T_i completes processing on I_j; all instances of the same type are assumed to process the task in the same amount of time
S	A schedule that maps tasks to instances that can process the tasks
U_{km}	Utility of a schedule using m instances of type k
Y_j	Type of instance I_j. Each type represents a certain configuration of CPU, memory, disk space and network speed.

Chapter 1

Introduction

1.1 Background on Cloud Computing

A recent trend in the IT industry has been the emergence of *cloud computing*, in which computing infrastructure is maintained in one or more large data centers and offered for use via a network connection [14]. The main advantages of such a setup are sharing of resources, economies of scale and flexibility to use as much or as little of the computing infrastructure as needed. This is in contrast to traditional computing infrastructure setups, where there is little flexibility and clients have to predict future computing needs well in advance. The predictions determine the infrastructure to purchase and are often incorrect, resulting in under-investment or over-investment in infrastructure. Other advantages of a cloud computing setup include instant access to a vast and varied amount of computing resources, flexible pricing where the client only pays for the resources used and avoidance of a single point of failure due to the way clients access the computing infrastructure and the redundancies built in to this infrastructure.

Cloud computing offerings can be classified into three broad types. The first type is Infrastructure-as-a-Service (IaaS), where only basic virtualized environments running on physical hardware are available. Everything from the operating system upwards must be installed and maintained by the client. This is useful when the client wants complete control over the software environment but does not want to deal with maintaining hardware or virtualized environments. The second type of offering is Platform-as-a-Service (PaaS), where the cloud service provider offers a virtualized environment with an operating system and common software required by most applications. The provider takes care of maintaining and updating this software, while the client focuses on building custom applications. The last type is Software-as-a-Service (SaaS), where entire software applications are offered on demand. In this setup, software applications are *rented* from the provider and are accessed over a network connection instead of being purchased and installed on the client's computing infrastructure. In all the above cases, the hardware is usually shared across multiple clients and access to hardware is abstracted away from the client.

Cloud computing offerings can also be classified as *private clouds* or *public clouds*. Private clouds are virtualized environments that are meant for the use of a single client and are accessed via a secure, private network connection. They cannot be accessed by anyone else. A public cloud, in contrast, is owned and operated by a cloud service provider and can be utilized by anyone who wants to rent infrastructure or software on demand. It is accessed via a public network like the internet. Examples of public clouds include Amazon Web Services from Amazon.com, Inc., Azure from Microsoft, Inc. and Google Cloud Platform from Google, Inc. This report deals only with PaaS offerings from public clouds, though the findings are equally applicable to private clouds.

The availability of a large amount of computing resources on demand and the ability to pay for these resources by the hour have enabled the migration of many workloads to a cloud computing environment [17]. These workloads can be processed in parallel on multiple resources and can complete much quicker than if they were processed on a single resource. While there is no practical constraint on the number of resources that can be used, budgetary constraints usually limit the number and type of resources that can be used to process a workload. Thus, there is always a trade-off between the benefit arising from processing a workload quickly and the cost of resources used to process the workload.

In most cases, the benefit is a fixed value and is paid out when all tasks in the workload are completed by a given deadline. It does not increase if the tasks are completed earlier and will not be paid out if tasks are not completed by the deadline. The only way for a client to maximize its profit is to complete all tasks by the deadline with the lowest possible processing costs. The rest of this report deals with developing efficient methods of scheduling tasks in parallel on multiple resources to achieve this goal.

1.2 Background on Scheduling

Scheduling refers to the process of assigning tasks to resources that complete the tasks. It is a common problem and arises in a variety of everyday situations, such as scheduling manufacturing tasks in factories, takeoffs and landings of planes in airports, sports tournaments, nurse shifts in hospitals and tasks across multiple computers. [13].

The input to a scheduling problem is a set of tasks to be completed and a set of resources that can process the tasks. The output is a schedule that maps tasks to resources in a certain order. Scheduling problems are typically subject to various feasibility constraints that define a valid schedule. Examples of such constraints include maintaining minimum time intervals between plane takeoffs and landings, avoiding adjacent shifts for nurses, completing tasks by a certain deadline and using a fixed number of machines. Once a set of feasible schedules is available, various optimization criteria are used select the best schedule from this set. Examples of such criteria include minimizing total weighted delay over all flights, preference of nurses for certain shifts and minimizing maximum completion time for tasks.

Scheduling problems can be classified in various ways based on task characteristics, resource characteristics and the type of optimization criteria used [13, 16]. Task characteristics include pre-emption, release times, deadlines, dependencies, weights and amount of pre-processing.

Pre-emption refers to the interruption of a task in progress so that it can be re-started later on the same or different resource. *Pre-emptive* schedules allow tasks to be interrupted while *non-pre-emptive schedules* do not allow pre-emption: once a task has started processing on a certain resource, it must complete processing on the same resource.

Tasks can be associated with release times and deadlines. Release time for a task represents the earliest time a task is available for processing. A task cannot start processing before its release time. A deadline represents the time by which a task must complete processing. When a set of tasks is associated with a release time, none of the tasks can begin processing before the release time. Similarly, when a set of tasks is associated with a deadline, all tasks in the set must complete processing by the deadline.

Dependencies between tasks control the order in which tasks can be processed. Examples of dependencies include precedence constraints between tasks, where one task depends on other tasks and cannot start processing until all tasks on which it depends have completed. Task weights and the amount of pre-processing tasks have received also influence the types of schedules that can be generated. Tasks with higher weights must usually be processed first and tasks that have been pre-processed typically require less processing time.

Scheduling problems can also be classified by the properties of resources. Tasks can be processed on a single resource or by multiple resources in parallel. When using resources in parallel, the resources can be *identical* or *unrelated*. Identical resources are all of the same type; task runtime will be the same regardless of which resource is used to process the task. When using unrelated resources in parallel, runtime for a task depends on the resource used to process the task and might vary for each resource.

Finally, scheduling problems can be classified by the the type of optimization criteria used to select the best schedule from a set of feasible schedules. Examples of such criteria include minimizing the maximum completion time (also known as *makespan*) of a set of tasks; minimizing the weighted completion time of a set of tasks where each task is assigned a weight proportional to its benefit; and minimizing the maximum delay of a set of tasks, if the tasks are associated with a deadline.

In this report, I focus on the problem of finding optimal non-pre-emptive schedules that minimize makespan while processing a set of tasks on multiple identical resources in parallel. All tasks have the same weight, have not undergone any pre-processing and are immediately available for processing.

1.3 Problem Formulation

The previous sections highlight the need for efficient scheduling methods to complete a set of tasks on time and with the lowest possible processing cost. This section specifies the problem in detail and suggests a possible solution. Subsequent sections develop this solution and show that it solves the problem in an efficient manner.

Let $\mathcal{T} := T_1, T_2, \ldots, T_n$ be the set of input tasks that must be processed. This set of tasks is associated with a benefit B specified in dollars and a deadline D specified in seconds. Each task T_i is independent of all other tasks in $\mathcal{T}$, so tasks can be processed in any order. $\mathcal{T}$ is considered to complete when all tasks in $\mathcal{T}$ are complete.

Let $\mathcal{I} := I_1, I_2, \ldots, I_m$ be the set of resources (also known as instances in this report) available to process tasks in $\mathcal{T}$. These instances are grouped into various instance types depending on their configuration. Let r_{ij} be the runtime of task T_i on instance I_j. Time taken to complete a task depends on the type of instance on which the task is processed; slower instances will take more time to process the same task compared to faster instances. Let C_j be the cost per hour of instance I_j (specified in dollars), with slower instances costing less than faster instances.

Let S be a schedule and let t_{I_j} be the index of the tasks in $\mathcal{T}$ assigned to instance I_j by S. The total runtime R_j for all jobs assigned to instance I_j is called the *load* on instance I_j and is defined as

$$R_j = \sum_{k \in t_{I_j}} r_{kj} \tag{1.1}$$

The maximum runtime across all instances, or the *makespan*, M of S is then defined as:

$$M = \max_{j \in \mathcal{I}} R_j \tag{1.2}$$

The makespan of a schedule represents the time taken by the schedule to complete all tasks and determines if the tasks will be completed by the deadline. A *feasible* schedule is defined as a schedule with $M \leq D$.

Makespan depends on the number of tasks, the number of instances and the type of instances used to process tasks. The number of tasks is an input to the problem, so the only parameters that can be varied are the number and type of instances. Makespan will be lowest when each task is processed in parallel on a separate instance of the fastest instance type and highest when all tasks are processed sequentially on a single instance of the slowest instance type. Processing cost depends on the number and type of instances used and the amount of time these instances were used.

To find a feasible schedule with the lowest processing cost, I need to evaluate schedules on a variety of instances types and vary the number of instances used in each case. A schedule can use any instance type whose configuration satisfies the minimum requirements of the tasks. The number of instances that a schedule can use can vary from 1

to the number of input tasks. I simplify my situation by assuming that a schedule can only assign tasks to instances of the same type, i.e., tasks can only be assigned to, say, 5 instances of a certain type, not 3 instances of one type and 2 instances of a different type. This constraint leads to the identical parallel machines scheduling problem. It has the advantage of restricting my search space and speeding up the search for the optimal schedule but the disadvantage of excluding schedules that use multiple instance types and result in lower makespans.

Section 1.2 mentions that the set of feasible schedules is first enumerated and the optimal schedule is then selected from this set. To obtain the set of feasible schedules, the makespan for all possible schedules must be calculated and compared to the deadline to determine feasibility. But the number of possible schedules is typically very large. The *Bell number* [18] refers to the number of ways a set with n elements can be partitioned into subsets such that each element is present in exactly one subset. Here each partition of the input set of tasks refers to a possible schedule. A set of 10 tasks can be assigned to instances into $115,975$ ways, with the number of instances varying from 1 to 10. 20 tasks result in 50×10^{12} schedules on 1 to 20 instances and 100 tasks result in 4×10^{115} schedules on 1 to 100 instances. Thus, even with a small number of tasks, the number of possible schedules is too large to exhaustively evaluate for feasibility. So it is not possible to first enumerate the subset of feasible schedules and then select the schedule with the lowest processing cost.

To work around this, I use *stochastic optimization* methods to find feasible schedules while simultaneously keeping track of their processing cost. Starting from an initial feasible schedule, I keep track of its processing cost and make it the first candidate solution. Stochastic local search methods are used to find other feasible schedules in the neighborhood of the current schedule that might have a lower processing cost. I continue the search in this manner until the termination criteria are met. When the search terminates, the feasible schedule with the lowest processing cost is the optimal schedule. A feasible schedule may not be found if the deadlines for the set of tasks are unrealistic. Even if the deadlines are realistic and a feasible schedule does exist, the benefit associated with a set of tasks may be lower than the lowest possible processing cost for the tasks and it will be more beneficial to not process the tasks than to process the tasks and incur a loss. To avoid these situations, I assume that the deadlines are always realistic and the benefit is higher than the lowest possible processing cost.

To evaluate feasible schedules, I adopt a *decision-theoretic* framework and *maximize expected utility* [3] to select the best schedule. One possible *utility* function evaluating a schedule using m instances of type k can be expressed as:

$$U_{km} = B - \left(\sum_{j=1}^{m} R_j \right) \cdot C_k \tag{1.3}$$

where U_{km} is the utility of a schedule using m instances, each of type k, B is the benefit derived from processing all tasks by the deadline D, C_k is the cost per hour of

an instance of type k, and R_j is the total time used by instance I_j of type k, rounded up to the next hour. B, D and C_k are fixed and known in advance. Note that both terms on the right-hand side of equation (1.3) are specified in the same units: B in dollars, C_k in dollars per hour and R_j in seconds (which when converted to hours and multiplied by C_k gives dollars).

Since all schedules have the same benefit, (1.3) simplifies to:

$$U_{km} = -\left(\sum_{j=1}^{m} R_j\right) \cdot C_k \tag{1.4}$$

Expected utility is maximized when the term of the right in (1.4), i.e., processing cost, is as small as possible.

An implicit assumption in (1.4) is that the task-level runtimes r_{ij} and instance-level runtimes R_j are deterministic and known in advance. But this is not always the case. It is not uncommon to find situations where the runtime of a task is known only when the task completes processing. In some cases, the runtimes are distributed according to a probability distribution that is known in advance. Each runtime can be distributed according to a different known probability distribution. In other cases, even the form of the probability distribution is not known or has no closed form expression. In such cases, the distribution of task runtimes must be determined empirically as described in the next section.

When runtimes are stochastic, (1.4) becomes:

$$E(U_{km}) = -\left[\sum_{j=1}^{m} \sum_{k \in t_{I_j}} E(r_{kj})\right] \cdot C_k \tag{1.5}$$

where t_{I_j} is the index of the tasks in $\mathcal{T}$ assigned to instance I_j by S. $E(U_{km})$ is the expected utility of a schedule using m instances of type k and I maximize *expected* utility by averaging over all possible runtimes.

Schedules for tasks with stochastic runtimes have stochastic makespans that have their own probability distribution. In this case, I relax my definition of feasibility so that any schedule with a makespan whose 95^{th} percentile is $\leq D$ is considered feasible. A major contribution of the current report is a method to find a feasible schedule with the lowest processing cost when tasks have stochastic runtimes that are distributed according to known or unknown probability distributions.

1.4 Data

The data for this problem is as follows:

- **Benefit**: Benefit is paid out only when all tasks are completed by the deadline. Specified in dollars

- **Deadline**: Time by which all tasks must complete processing. Specified in seconds.

- **Instance types:** Each Amazon data center has hundreds of thousands of virtual machines (also known as instances) [2] running on physical hardware. Each instance is associated with a certain configuration of processor type and speed, memory, local disk space and network speed. This configuration represents an instance type. Examples of instance types include compute-optimized instance types with faster, more powerful processors, memory-optimized instance types with large amounts of memory and storage-optimized instance types with large amounts of local disk space. There are dozens of such instance types and thousands of instances of the same type in each data center. Instance types and the number of instances of a certain type represent the action space of the decision problem. In this report, I assume that tasks are constrained only by processor speed and not by any other properties of an instance.

- **Instance costs**: Each instance type is associated with an hourly cost that is determined by Amazon and specified in dollars.

- **Task lengths**: The input to the problem is a set of tasks. Each task is associated with a property called *length*. Runtime for a task on a given instance type is roughly proportional to its length. Besides length, other properties of the task also influence task runtime. If these properties were easy to quantify, then runtime would be a deterministic function of length and the other properties. But these properties are not easily quantifiable, so task runtime is considered to be a random unknown with length as the only co-variate that can be used to predict runtime on the given instance type.

Chapter 2

The single instance case

This chapter focuses on selecting optimal feasible schedules when processing tasks on a single instance, and the next chapter deals with the case of processing tasks on multiple identical instances in parallel.

2.1 Method

Equation (1.2) defines the makespan of a schedule as the maximum load across all instances used by the schedule. When a schedule processes tasks on a single instance, there is only one possible schedule and the makespan of this schedule is the same as the load on the given instance. This makespan is invariant to the order in which tasks are processed on the instance. Since the number of instances is fixed at 1, the action space is reduced to the set of instance types that result in feasible schedules, and the action is the selection of the instance type from this set that maximizes (expected) utility.

When task runtimes are deterministic and known in advance, the makespan M of a schedule S using instance type k is just the sum of the runtimes for k. If $M \leq$ deadline D, then S is feasible and the utility U_{k1} is calculated for these instance types as shown in equation (1.4) with $m = 1$. This process is repeated for each instance type in the action space and the feasible schedule with the maximum utility is the solution to the problem. All tasks should be processed on the instance type associated with this schedule.

When runtimes are stochastic, M, which is the sum of runtimes, is itself random with an associated probability distribution. The distribution of M depends on the nature and number of runtime distributions being summed. Tasks in $\mathcal{T}$ are assumed to be independent and represent a random sample from the population of tasks. The same applies to the runtimes of these tasks. If the number of tasks or, equivalently, runtimes, is a large enough number, say, 100, then the Central Limit Theorem (CLT) applies and M will have a *Normal* distribution. The mean and variance of the Normal distribution will equal the sum of the means and variances, respectively, of the individual runtime distributions. If the number of tasks is not large enough for the CLT to apply, then

the distribution of M depends on the nature of the distributions being summed. If the runtimes all have distributions that are members of the same family, it is possible for M to have a distribution of this family whose parameters are a simple function of the parameters of the individual runtime distributions. For instance, if the individual runtimes distributions are all Poisson distributions, then M will also have a Poisson distribution whose parameter is the sum of the parameters of the individual Poisson distributions.

If the runtime distributions are not members of the same family or the distribution of M cannot be determined easily from the individual distributions, then the distribution of M can be determined via bootstrap resampling [5]. A value is randomly sampled from the runtime distribution of each of the input tasks. These values are then summed to give a value for makespan. This process is repeated a large number of times, resulting in a bootstrap distribution for makespan.

If the runtime distribution for one or more tasks is unknown or does not have a closed form expression, a value must be sampled from the empirical distribution for the runtime. The empirical runtime distribution for a task with length l on an instance of type k is determined by processing a number of samples of tasks with length l on an instance of type k and recording the runtimes. This set of runtimes represents the empirical probability distribution for the runtime and can be sampled to get a value for the runtime of the associated task.

In all the above cases with stochastic runtimes, the 95^{th} percentile of the makespan distribution is assumed to represent the makespan of the input set of tasks on the instance type under consideration. This value is used to calculate expected utility and determine the instance type that maximizes expected utility.

2.2 Validation

When runtimes are deterministic and known in advance, determining the makespan is straightforward. Table 2.1 shows the details for a set of 5 tasks associated with a benefit of \$30 and a deadline of 25 hours. These tasks can run on 3 different instance types A, B, and C. The instance types vary in their cost and speed of processing, with the cheaper instance being slower than the more expensive instances. Makespan is rounded up to the next hour since I pay for the entire hour even if I use only a fraction of the hour. Only B and C result in feasible schedules (makespan $\leq$ deadline) with C having the maximum utility even though it has the highest cost per hour. The schedule using instance type C is the optimal schedule for the given set of tasks.

2.2.1 ≥ 100 tasks/instance

When 100 or more input tasks with stochastic runtimes are submitted, the makespan has a Normal distribution by the Central Limit Theorem (CLT). (100 is an arbitrary threshold chosen to satisfy the 'sufficiently large number of values' assumption of the

Instance type	**Relative Speed**	**Cost ($/hr)**	**Runtimes (hr)**	**Makespan (hr)**	**Utility ($)**
A	1	1	5.2, 8.5, 2.5, 4.1, 6.0	27	—
B	1.07	1.1	4.84, 7.91, 2.33, 3.81, 5.58	**25**	3.09
C	1.35	1.3	3.85, 6.29, 1.85, 3.03, 4.44	**20**	**4.7**

Table 2.1: *Calculating utility for a set of tasks whose runtimes are deterministic and known in advance. The benefit is $30 and the deadline is 25 hours.*

CLT). To validate the above method, tasks are chosen whose actual runtimes are known. These runtimes represent the 'truth' and are used to calculate the true optimal schedule using the method for deterministic runtimes. This true optimal schedule is then compared to the predicted optimal schedule to determine the effectiveness of the above method.

Each schedule is associated with an instance type, makespan and processing cost. Processing cost is a constant times the makespan, so minimizing makespan on the optimal instance also minimizes the processing cost. I consider a predicted optimal schedule to be equivalent to the true optimal schedule if the instance types of both schedules are identical and the 95% confidence interval for predicted makespan contains the true makespan. This confidence interval is based on a Normal approximation to the makespan distribution. 95% is again an arbitrary threshold and was chosen since it is considered to be 'close enough' to the optimal makespan by most people.

I decided to validate collections of tasks containing 100, 250, 500 and 1000 tasks. 1000 collections of tasks were generated for each of the above sizes (e.g., 1000 sets of 100 tasks each). For each task in each collection, runtime was sampled from one of four distributions with positive support (Poisson, Exponential, Gamma and Uniform). The parameters for these distributions were arbitrary and could be different for different tasks. For each task, the distribution and parameters used were recorded, in addition to the sampled value. So 100 tasks in a collection, for instance, could contain runtimes from all 4 distributions in different quantities. The true optimal schedule (instance type and makespan) was calculated for each collection of tasks using the sampled values assuming an action space of the 3 simulated instance types (A, B and C) mentioned in the previous section. The parameters of the Normal distribution for the makespan associated with each instance type were determined from the parameters of the individual distributions of runtimes. The optimal schedule was the one that completed all tasks with the lowest cost and with a makespan that was within 5% of the deadline.

Figure 2.1 shows the results of these predictions when processing 100 tasks at a time on a single instance of a given instance type. Each black point represents the true makespan for a set of 100 tasks. There are 1000 such points in the plot. The red dots represent the 95% confidence interval for predicted makespan based on a Normal approximation to the makespan distribution. 94.7% of the predicted schedules used the optimal instance type while the remaining 5.3% used a sub-optimal instance type.

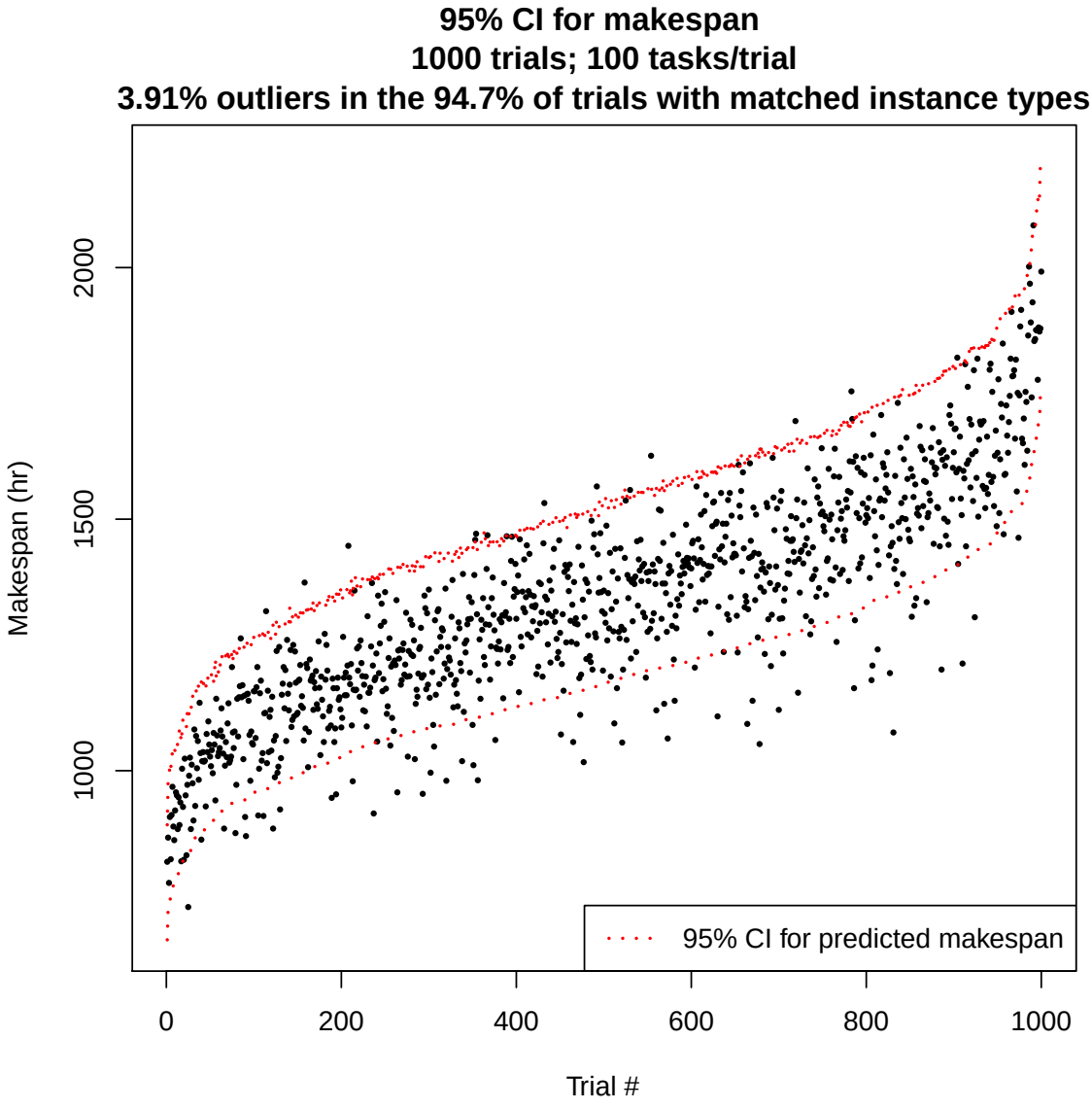

Figure 2.1: *95% confidence interval for makespan using Normal approximation when 100 tasks are processed on a single instance. Data sorted by lower bound of 95% CI.*

Of the predictions using the optimal instance type, the 95% confidence interval for predicted makespan contained the true value in all but 3.91% of the cases. So the 95% CI for predicted optimal schedule effectively contained around 91% of the true optimal schedules. Figures 2.2, 2.3 and 2.4 show the corresponding results when processing 250, 500 and 1000 tasks/instance respectively on the same 3 instances types. In these cases, the method performs better and the confidence intervals contain almost 95% of the true optimal schedules. When the runtime distributions are unknown, runtimes are sampled from the corresponding empirical distributions. Results in this case are very similar and are not shown.

2.2.2 < 100 tasks/instance

When an instance is assigned less than 100 tasks/instance, the Central Limit Theorem does not apply and the makespan distribution must be determined through other means. I use bootstrap re-sampling to construct a distribution for the makespan. Simulated collections of tasks are generated in the same manner as in the previous section and the predicted optimal schedules are compared to the true optimal schedules.

Figure 2.5 shows the 95% confidence interval for makespan when 10 tasks are processed on a single instance. In this case, the predictions are not very good - only 77% of the predictions use the optimal instance type. Of this 77%, the 95% confidence interval contains all but 2.08% of the true values of the makespan. Figures 2.6, 2.7 and 2.8 show the corresponding confidence interval for 25, 50 and 75 tasks respectively. The results here are much better - almost all the predictions use the optimal instance type and the number of predictions outside the 95% confidence interval is close to the error tolerance level of 5%. So the total percentage of incorrect predictions is also close to 5%. The results suggest that predicting optimal schedules using bootstrap approximation is applicable only when at least 25 tasks are assigned to an instance and cannot be used when processing fewer number of tasks per instance. Results (not shown) when sampling from empirical distributions for runtimes, rather than distributions with closed form expressions, are also very similar.

2.3 Summary

In this chapter I used two different methods to predict the optimal schedule when processing a set of tasks on a single instance. If the number of tasks is ≥ 100, the makespan distribution is approximated by a Normal distribution which is then used to predict optimal instance type and makespan. Prediction results are good and the percentage of incorrect prediction is around the error tolerance of 5%. If the number of tasks is < 100, I use a bootstrap approximation to the makespan distribution and then predict the optimal schedule. Results using this method are poor when using ≤ 25 tasks but improve to expected levels when processing > 25 tasks. These methods form

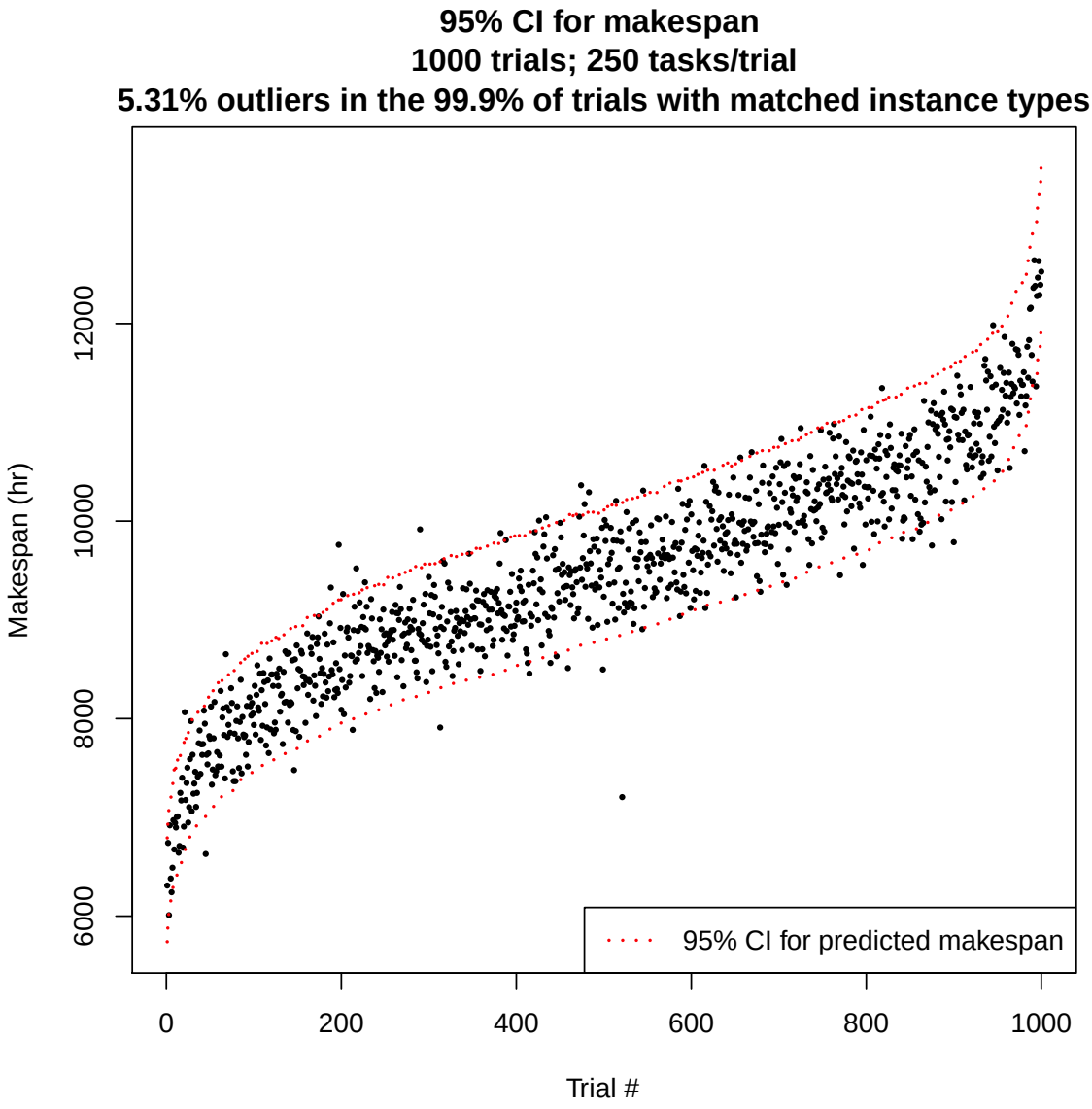

Figure 2.2: *95% confidence interval for makespan using Normal approximation when 250 tasks are processed on a single instance. Data sorted by lower bound of 95% CI.*

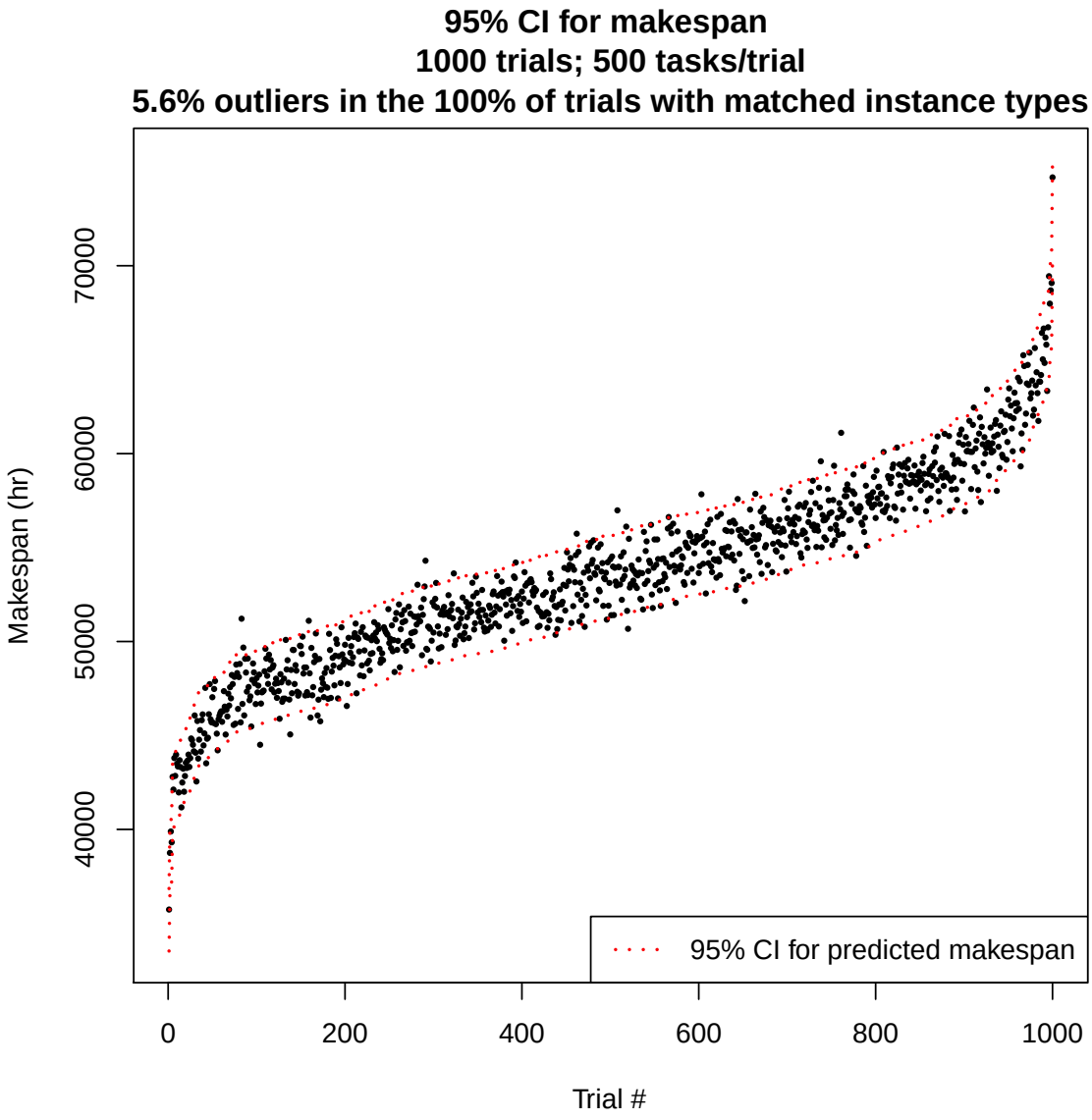

Figure 2.3: *95% confidence interval for makespan using Normal approximation when 500 tasks are processed on a single instance. Data sorted by lower bound of 95% CI.*

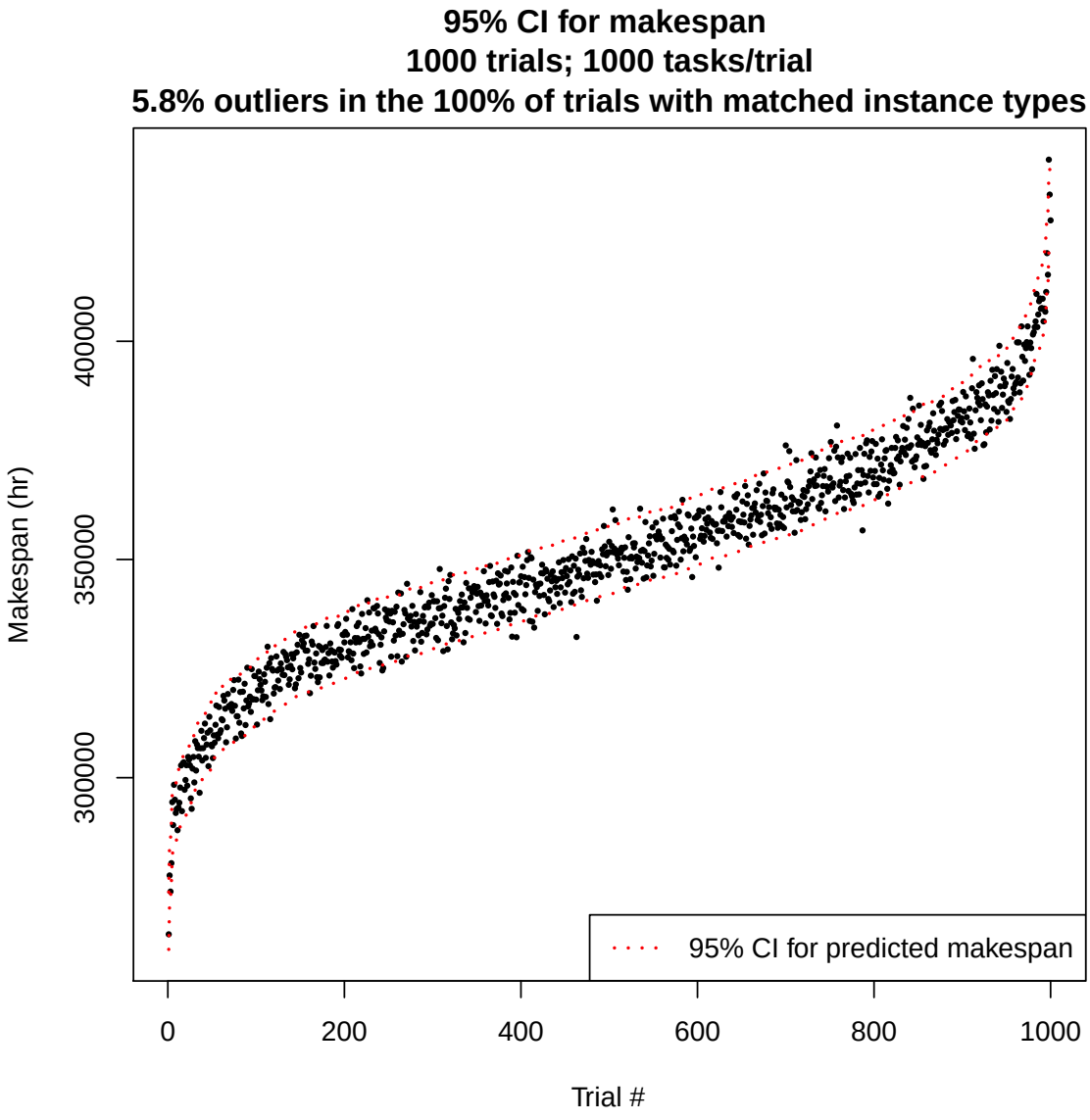

Figure 2.4: *95% confidence interval for makespan using Normal approximation when 1000 tasks are processed on a single instance. Data sorted by lower bound of 95% CI.*

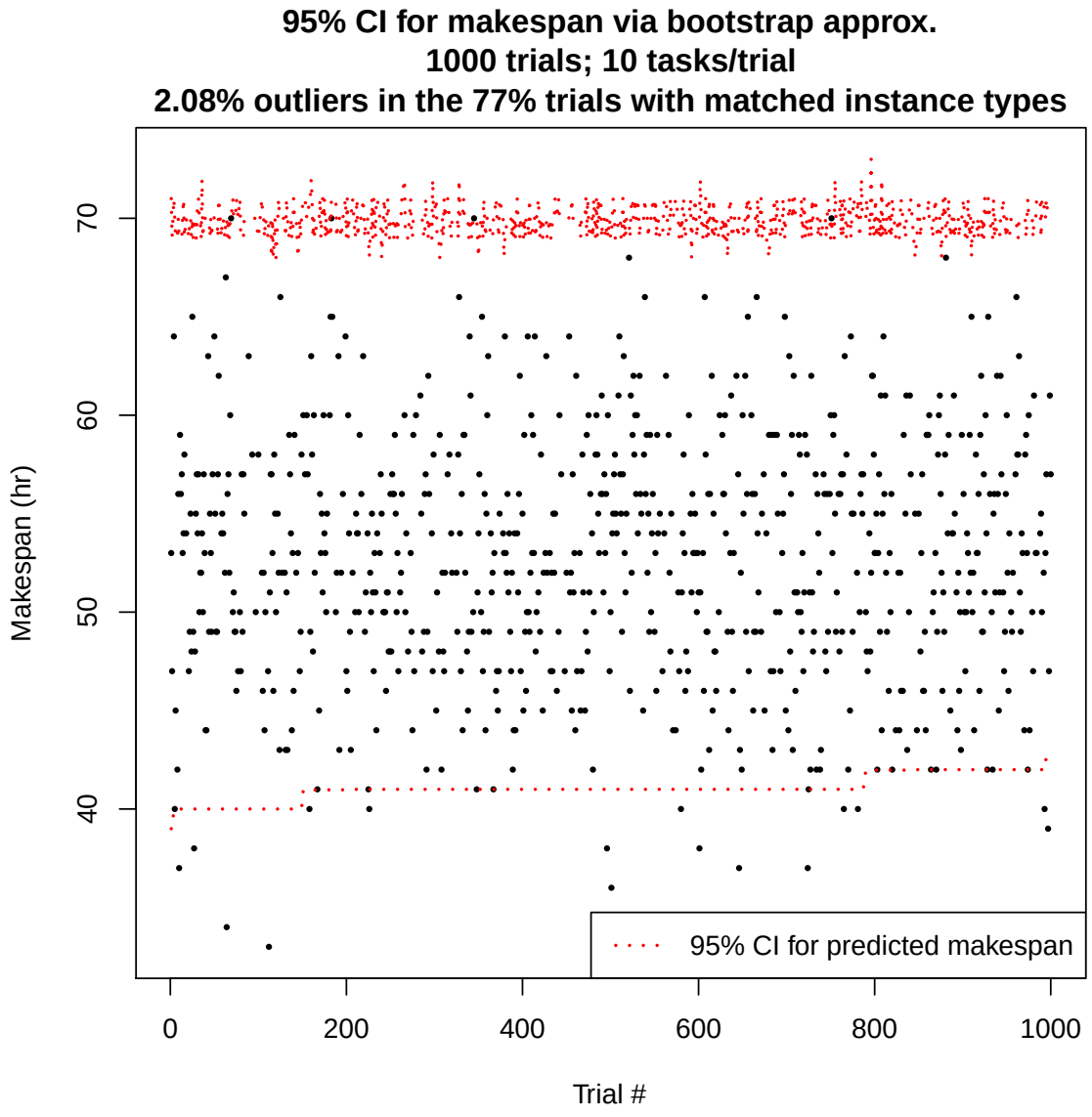

Figure 2.5: *95% confidence interval for makespan using Bootstrap approximation when 10 tasks are processed on a single instance. Data sorted by lower bound of 95% CI.*

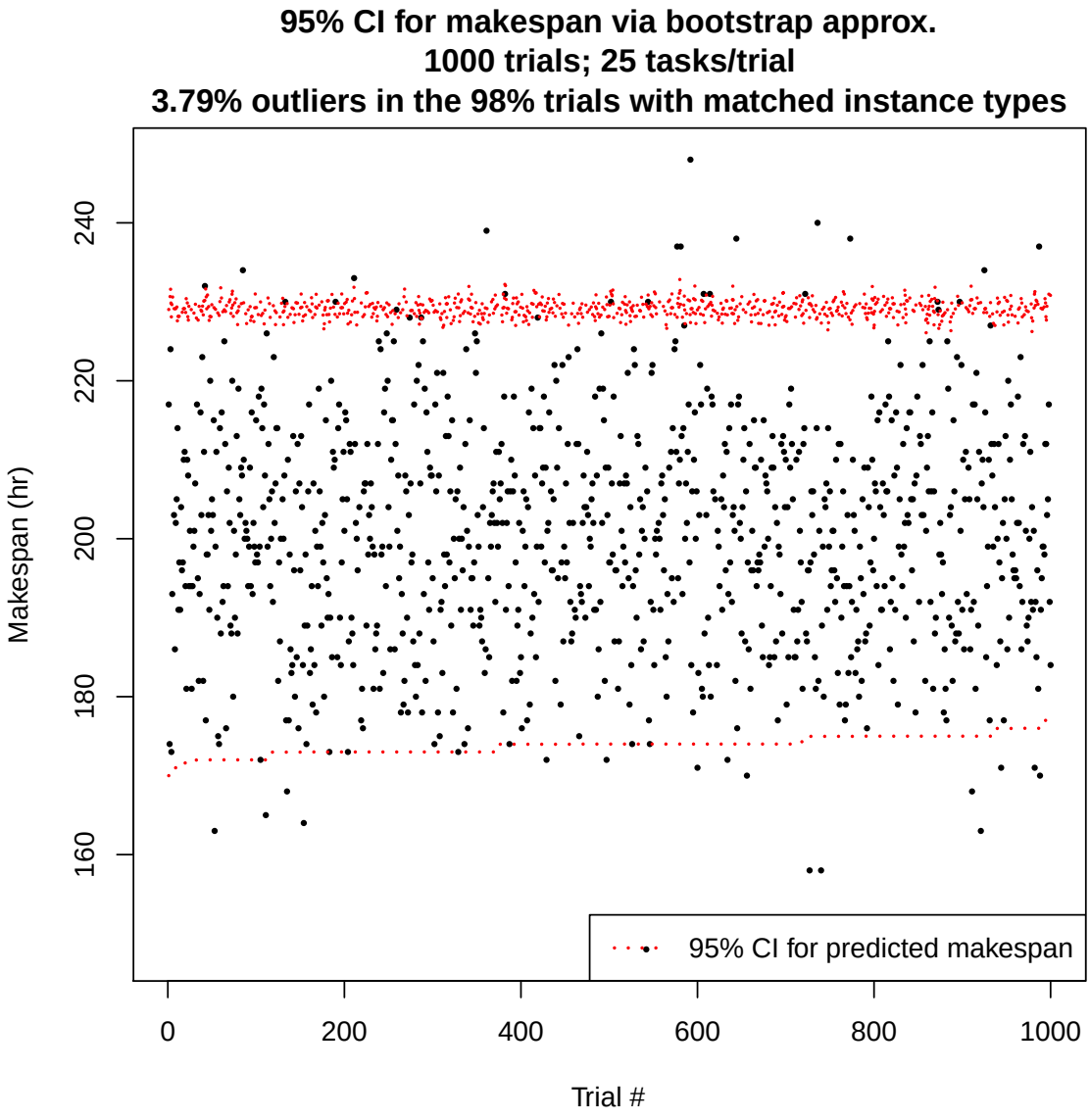

Figure 2.6: *95% confidence interval for makespan using Bootstrap approximation when 25 tasks are processed on a single instance. Data sorted by lower bound of 95% CI.*

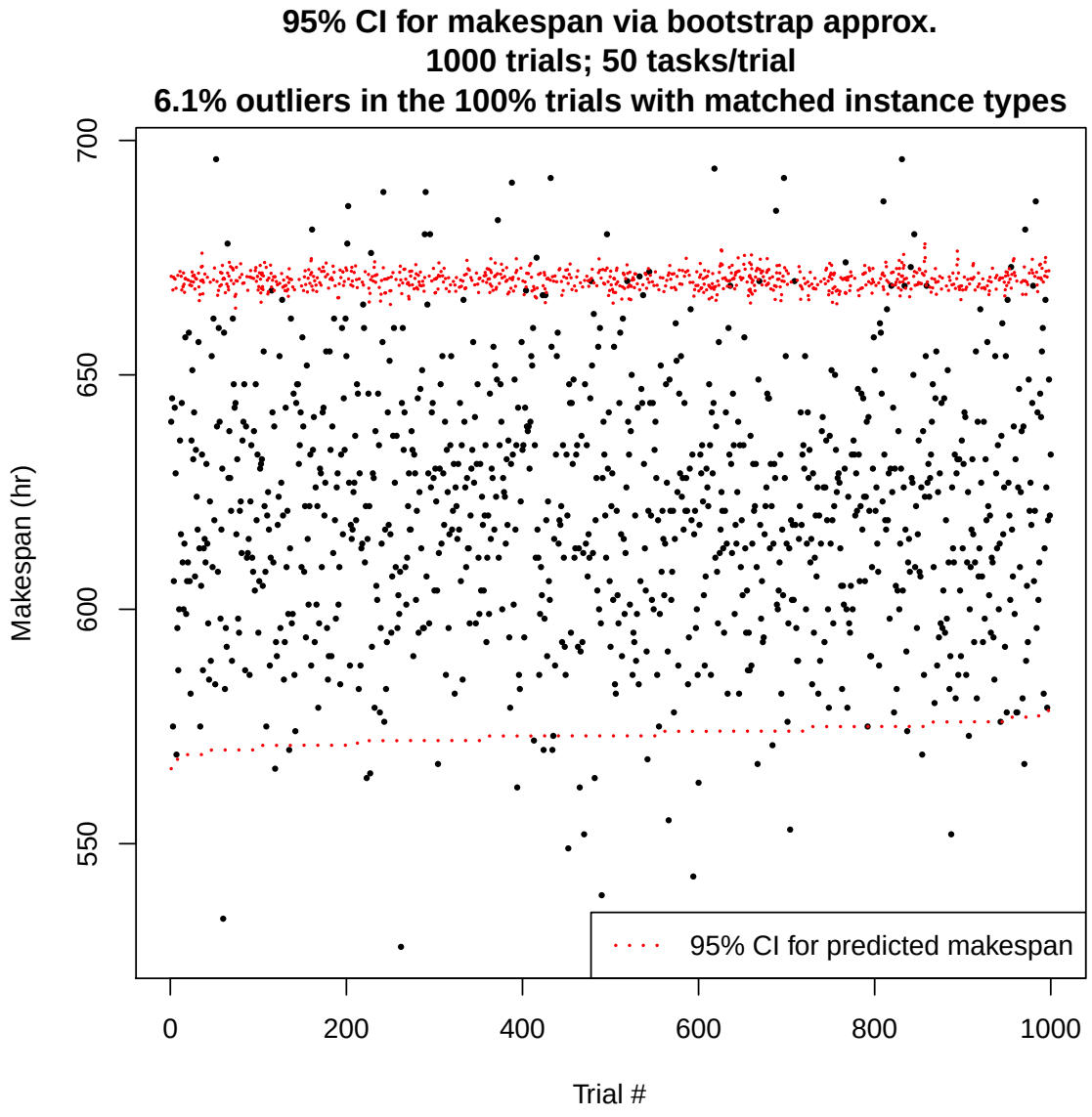

Figure 2.7: *95% confidence interval for makespan using Bootstrap approximation when 50 tasks are processed on a single instance. Data sorted by lower bound of 95% CI.*

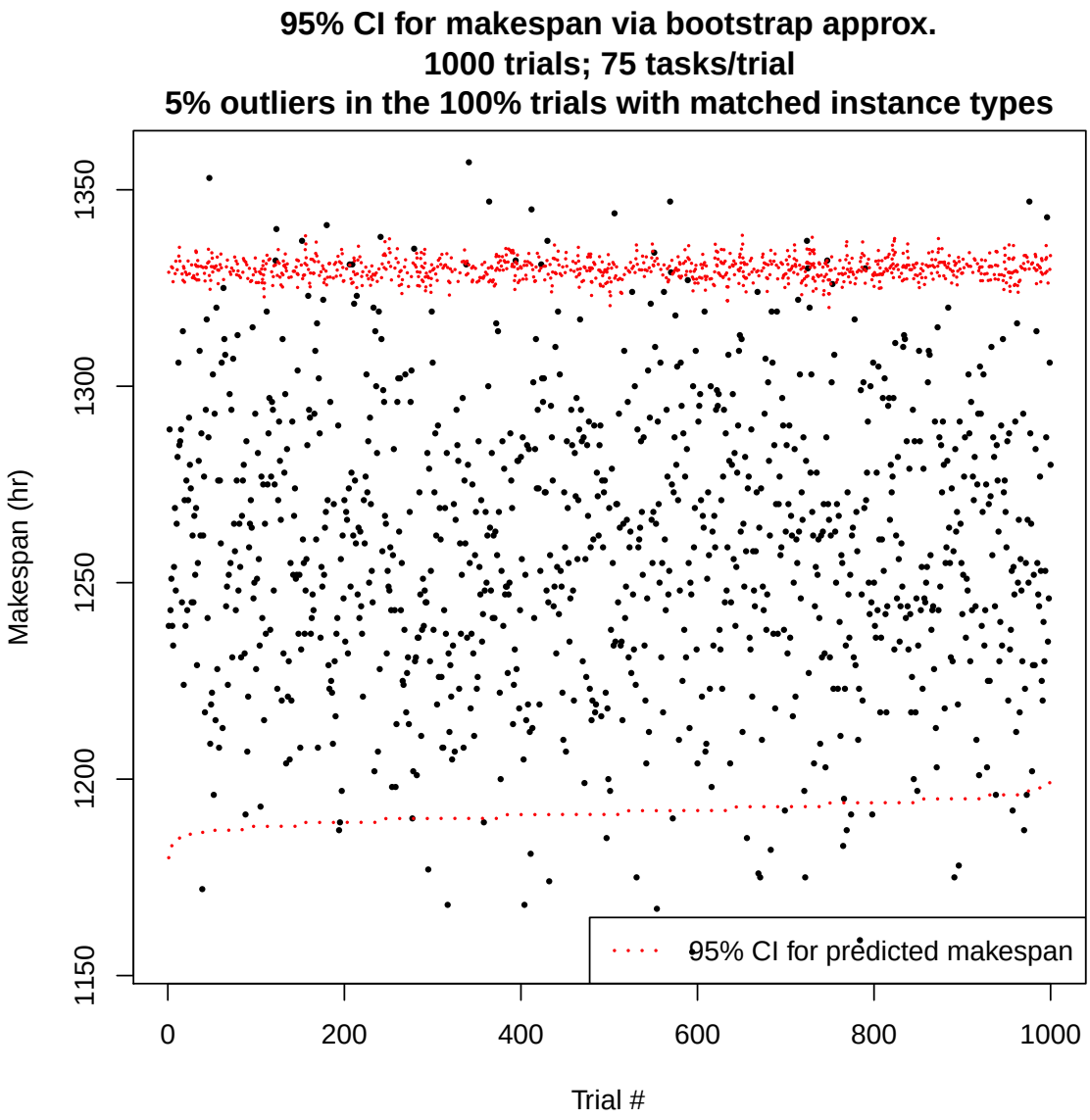

Figure 2.8: *95% confidence interval for makespan using Bootstrap approximation when 75 tasks are processed on a single instance. Data sorted by lower bound of 95% CI.*
ha

the core of the method used in the next section when scheduling tasks across multiple instances.

Chapter 3

The multiple instance case

When scheduling tasks on single instances with the objective of minimizing makespan, there is only one possible schedule for each instance type. When scheduling tasks on multiple instances, however, the number of possible schedules is typically very large and it is not possible to find the globally optimal schedule in a 'reasonable' amount of time. In fact, most scheduling problems, including the problem of minimizing makespan, are considered to be NP-hard [7] and form an important class of combinatorial problems.

A common algorithm to find a near-optimal schedule that minimizes makespan for deterministic runtimes when using multiple instances is called the Longest Processing Time First (LPTF) rule. When runtimes are stochastic and follow the Exponential distribution, the Longest Expected Processing Time First (LEPTF) rule gives a near-optimal schedule [16]. These rules first order all tasks in decreasing order of (expected) runtimes. They then assign the task with the largest (expected) runtime that has not yet started processing to the next available instance. The schedule obtained by these rules has a makespan that is at most $\left(\frac{4}{3} - \frac{1}{3m}\right)$ greater than the makespan of the optimal schedule, where m is the number of instances used by the schedule [11].

The proof of the near-optimality of the LEPTF rule relies on the memory-less property of the Exponential distribution, so this rule does not apply if the runtimes follow a different distribution. In such cases, local search methods are used to find, in a 'reasonable' amount of time, a schedule that is 'close enough' to the global optimum. Local search methods start with a candidate solution and explore all solutions in the *neighborhood* of this solution [8]. If a *better* solution is found, the search centers around the neighborhood of this solution. This will ensure that the search is always moving toward the global optimum. To avoid getting stuck in local optima, local search methods occasionally move to a *worse* solution and explore the neighborhood around that solution in the hope of finding a path to the global optimum.

Fouskakis [6] compared several local search methods while trying to find the optimal set of predictors by trading off prediction accuracy against the cost of predictors. Simulated annealing, genetic algorithms, tabu search and their variants were compared and simulated annealing was found to perform better than other methods. I chose simulated

annealing as the local search method in this report based on its performance and the simplicity of its implementation.

3.1 Method

The steps below describe the simulated annealing (SA) algorithm to find the optimal schedule when runtimes are distributed according to an arbitrary distribution:

- **Initialization:** Select an initial feasible schedule as the current solution and compute its processing cost.

- **Candidate generation:** Given a schedule, a neighboring schedule is generated by re-assigning one or more tasks between one or more instances used by the schedule. Task re-assignment means moving or exchanging tasks between one or more instances. The greater the proportion of tasks re-assigned or the greater the proportion of instances whose tasks are re-assigned, the more different the new schedule will be from the current schedule. The proportion of tasks re-assigned and the proportion of instances whose tasks have been re-assigned together represent a 'variance' parameter. Large values of 'variance' allow for the exploration of schedules very different from the current schedule while small values result in the exploration of the neighborhood around the current schedule. Ideally, we will want to use large values of 'variance' in the initial stages of the search and small values later in the search once a 'good' schedule has been found. In this report, we do not vary the 'variance' but used fixed values for the proportion of tasks re-assigned and number of instances between which the tasks are re-assigned. The values selected for the variance parameters attempt to strike a balance between exploring schedules that are very different from and those that are very similar to the current schedule. The number of instances between which tasks can be re-assigned is fixed at 2 and the maximum proportion of tasks assigned to an instance that can be re-assigned is fixed at $(1/3)^{\text{rd}}$.

  In each iteration, an instance used by the schedule is selected at random. The number of tasks assigned to this is instance is reduced by a third and rounded up to the next integer. This value represents the maximum number of tasks that can be re-assigned. A value is randomly sampled between 1 and this value. This is the actual number of tasks that will be re-assigned in this iteration. Then all instances containing at least this many tasks are selected. From this subset of instances, 2 instances are selected at random and tasks are moved or exchanged between these 2 instances. The decision to move or exchange tasks between the 2 instances is also made randomly in each iteration. If it is not possible to find 2 instances containing the number of tasks to be re-assigned, then one task is moved between 2 instances selected at random. By proceeding in this manner, in each iteration, a new candidate schedule is generated in the neighborhood of the

current schedule. Note that this process is independent of temperature (defined below) and works the same way in all iterations.

After a candidate schedule has been generated, its makespan is computed. For deterministic, known runtimes, compute makespan as shown in (1.1). For stochastic runtimes, makespan is the 95^{th} percentile of the makespan distribution that is determined using the Normal approximation or the bootstrap approximation as specified in Chapter 2. In both cases, compute makespan for each instance in the set of instances being used to process tasks. The makespan of the candidate schedule is equal to the maximum of the makespans of the individual instances (equation (1.2)).

- **Acceptance:** If the makespan of the candidate schedule is greater than the deadline, the schedule is not feasible, so reject it. Else, calculate the processing cost for the candidate schedule. If this processing cost is less than the processing cost of the current schedule, accept the candidate schedule. Else, accept the candidate schedule with a probability that is a function of the processing cost of the current schedule, the processing cost of the candidate schedule and the current temperature.

  Temperature is an important parameter of the simulated annealing algorithm and controls the move from the current schedule to the candidate schedule. The simulated annealing algorithm starts off with an initial temperature and this temperature is decreased as the algorithm progresses, mimicking the annealing process in metallurgy from which the algorithm gets its name. Initial higher temperatures allow moves to feasible candidate schedules with higher processing costs, resulting in exploration of different parts of the search space. As temperature decreases, the probability of moving to a schedule with a higher processing cost decreases and the algorithm tends to stay close to the most recently found best schedule.

- **Termination:** The algorithm terminates when the maximum number of iterations has been reached. The feasible schedule with the lowest processing cost found so far is the approximate solution to the problem.

By proceeding as above for a suitably large number of iterations, I explore the search space of possible schedules and find a feasible schedule that is 'close enough' to the globally optimal schedule. The number of iterations and the starting temperature are fixed in advance. A linear *cooling schedule* is used where the temperature decreases linearly with each iteration. There is only 1 iteration at each temperature. While exploring the search space, it is possible for the SA algorithm to make a series of bad moves and end up with a schedule that is considerably worse than the initial schedule. To prevent this, the algorithm moves back to the best schedule determined so far when there is a 10% increase in processing cost compared to the current best schedule. This helps a lot with improving the efficiency of exploring the search space. The algorithm assumes that the number of instances is fixed. It must be repeated for every combination

of instance types and cluster size that I want to compare to determine the optimal combination.

3.2 Validation

To validate the simulated annealing algorithm, I need a simulated data set where 'truth' is known. Here I focus on the case when runtimes are distributed according to an unknown distribution and an empirical distribution for runtimes must be obtained. As previously mentioned, each task is associated with a co-variate called length and the length can range from 1 to 3000. It is known that, in the general population, tasks with smaller length occur more often than tasks with larger lengths. I selected a set of 87 lengths that form a representative sample of the task length population (figure 3.1). Task runtimes are assumed to be exponentially distributed to facilitate comparison with results from the LEPTF rule mentioned in the previous section. Since runtimes are roughly proportional to length, the parameters of the exponential distributions used to generate the 'true' runtimes increase with length. Distributions of runtimes for tasks with a smaller length will be shifted to the left compared to those for tasks with larger lengths. For each of the 87 lengths, I generated a set of 200 samples from an exponential distribution. These samples represent the empirical distribution of runtimes for the set of input tasks. The SA algorithm only uses the empirical distributions and does not know that they are generated from exponential distributions.

As in chapter 2, I create a collection of input tasks and a set of instances that will process the tasks. In this validation setup, the number of instances is fixed at 2 (both of the same instance type), the number of tasks in the collection is fixed at 10 and the benefit and deadline are given. To create the collection of input tasks, I sampled with replacement 10 values from the set of 87 lengths. The LEPTF rule is used to determine the near-optimal schedule for processing these 10 tasks on the 2 instances. This schedule represents the 'truth' to which the prediction of the SA algorithm will be compared. The SA algorithm is also used to predict the optimal schedule for the processing the same set of tasks on the same 2 instances. The probability of completing the tasks by the deadline and the cost of processing the tasks are calculated for both schedules. This process is repeated for 100 sets of input task collections, each with 10 tasks, and the results are compared.

Figure 3.2 shows the probability of completing the tasks by the deadline. Note that all probabilities are rounded to 2 digits after the decimal point. Schedules generated by the LEPTF rule in all 100 trials have at least 95% probability of completing the tasks by the deadline. Most of the schedules generated by the SA algorithm also have a $\geq 95\%$ chance of completing the tasks by the deadline. But the probability for the schedules for a few trials is well below 95%. I attribute this to being unable to find a feasible schedule within the given number of iterations. This problem is usually remedied by letting the SA algorithm run longer.

Figure 3.3 shows the costs associated with the same schedules. Costs for schedules

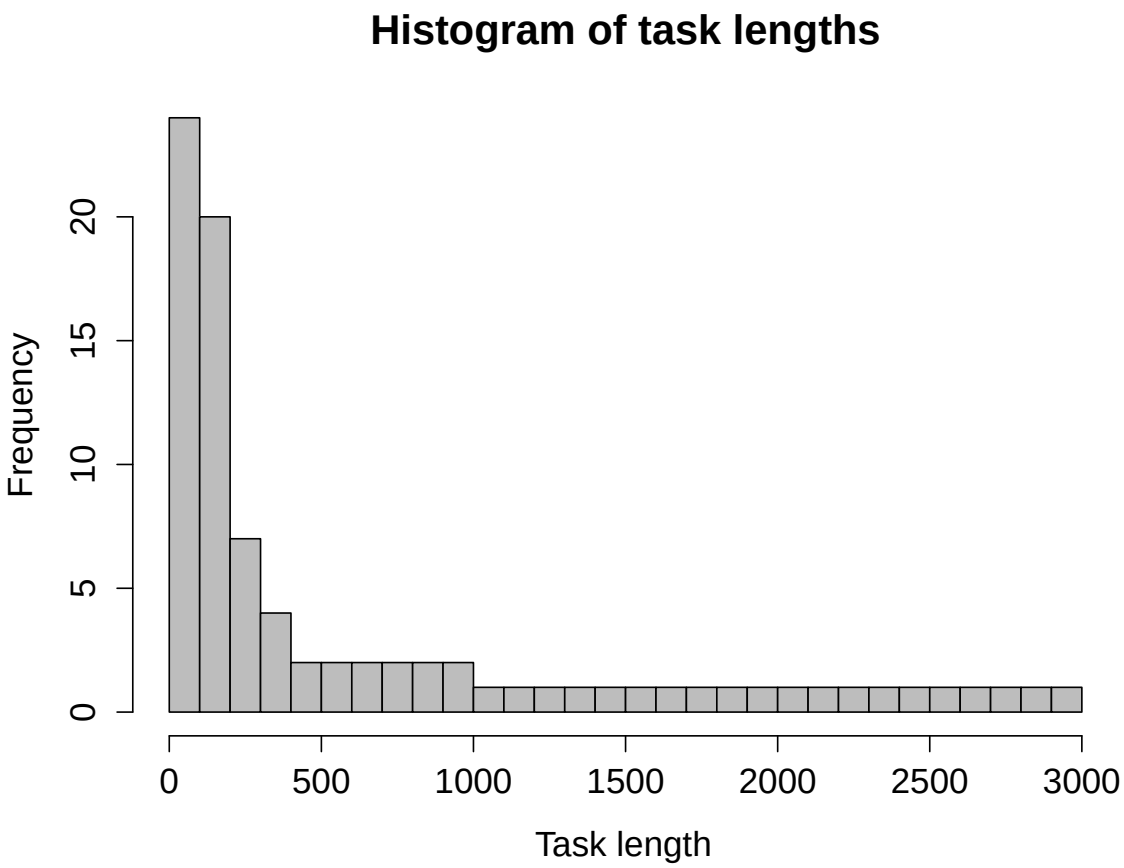

Figure 3.1: *Histogram of representative task lengths used for validation.*

generated by both methods are strongly correlated (Pearson correlation = 0.98). But costs for SA schedules are almost uniformly lower than costs for the LEPTF for the same set of tasks. This is unexpected since LEPTF generates the near-optimal schedule with the near-optimal cost while SA may not always generate the optimal schedule and I expect most of the points to be on or below the diagonal line. One reason for this behavior could be the size of the simulated data sets, each of which contains only 200 samples. Another reason could be that the SA algorithm is actively minimizing both makespan and cost while the LEPTF rule is minimizing only makespan.

The above validation study is limited in the sense that it used only a single cluster (1 instance type, 2 instances). The study can be easily extended by defining additional clusters by varying the instance type or number of instances or both and running the SA algorithm on each of these clusters. The cluster that results in the most optimal schedule should be the one used to process the input tasks.

3.3 Summary

This chapter described a method that uses Simulated Annealing to find the optimal feasible schedule for a set of tasks whose runtimes follow an unknown distribution. The algorithm was validated using exponentially distributed runtimes and the schedules generated by the algorithm were compared to the corresponding schedules generated by the LEPTF rule. The results were favorable with the percentage of sub-optimal schedules (i.e., schedules not completing the tasks by the deadline) being under the error tolerance of 5%.

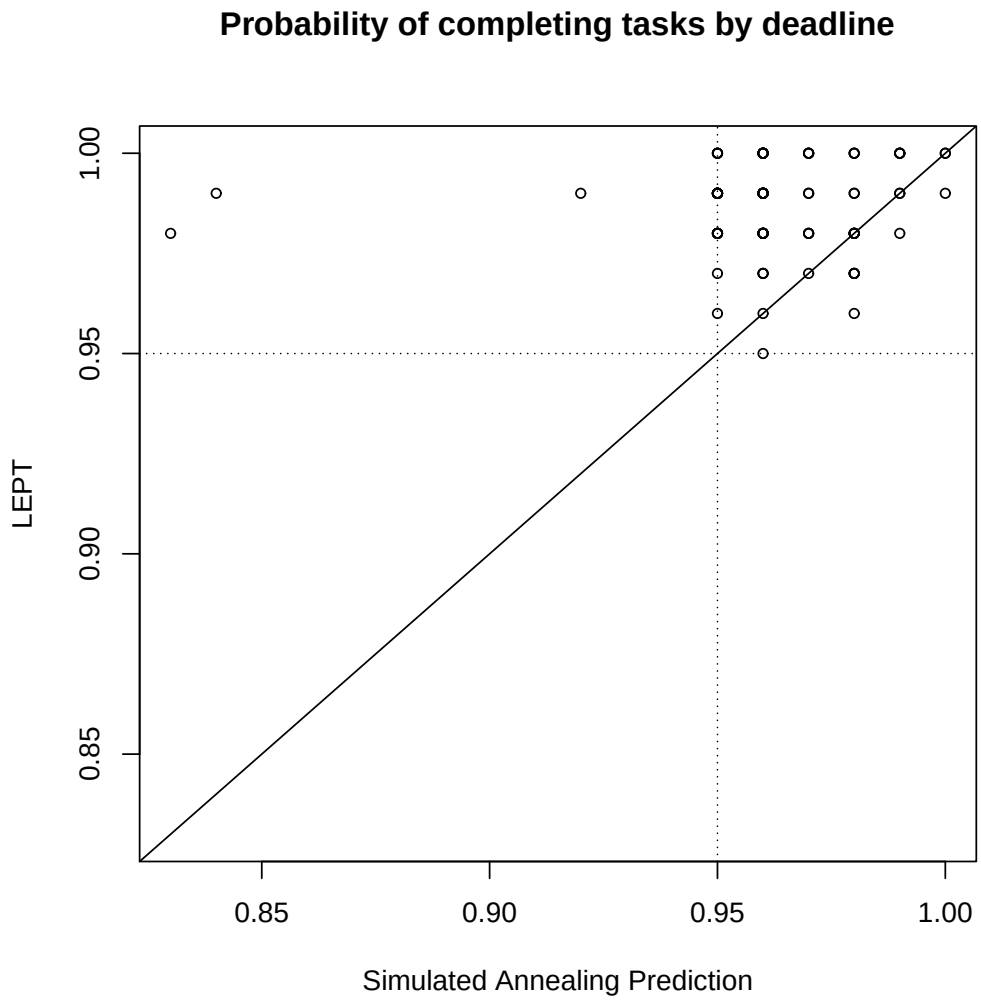

Figure 3.2: *Probability of completing tasks by the deadline for schedules generated by the Simulated Annealing scheduling algorithm and the Longest Expected Processing Time First (LEPTF) rule when runtimes are exponentially distributed.*

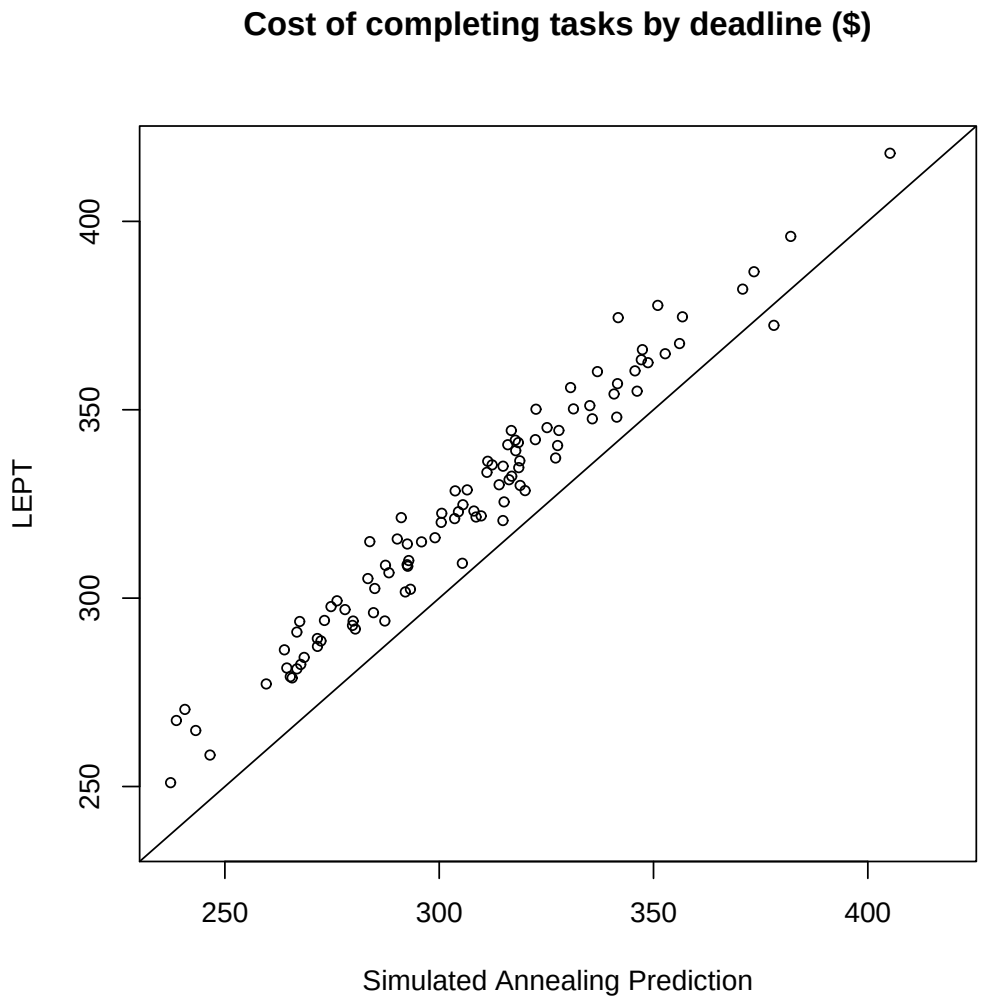

Figure 3.3: *Cost of completing tasks by the deadline for schedules generated by the Simulated Annealing scheduling algorithm and the Longest Expected Processing Time First (LEPTF) rule when runtimes are exponentially distributed.*

Chapter 4

Conclusion and future work

4.1 Conclusion

In this report I developed methods to find optimal schedules for tasks with stochastic runtimes. When processing tasks on a single instance, a Normal approximation or a bootstrap approximation to the makespan distribution was used depending on the number of tasks to be processed on the instance. The $95th$ percentile of this makespan distribution was used to determine the optimal schedule. The optimal schedule was predicted with approximately 95% accuracy for simulated data sets when at least 25 tasks were being processed on an instance. The percentage of sub-optimal predictions was around the error tolerance of 5%. When processing tasks on multiple instances, simulated annealing was used to explore the search space of feasible schedules. The results from the above method compared favorably to results from the LEPTF rule.

4.2 Future work

The methodology developed in the previous chapters can be extended in the following ways:

- **Varying SA parameters:** Simulated annealing has several parameters that can be tuned to the problem at hand. Only a few parameters are explored in this report. Other parameters worth exploring include improved methods of candidate generation, skipping evaluation of small clusters of slow instances that have no chance of completing all tasks by the deadline, alternative cooling schedules, multiple iterations at the same temperature and re-starting the annealing process from the starting temperature after moving back to a previous solution. Some of these options are discussed in Fouskakis [6].

- **Using alternative stochastic local search methods:** The simulated annealing algorithm [15] used in Chapter 3 for finding the optimal schedule is just one

member of a class of algorithms known as meta-heuristics for solving combinatorial problems. Other members of this class include tabu search [9, 10], genetic algorithms [12], Ant Colony Optimization [4] and their variants [13]. Performing a comparison similar to the one done in Fouskakis [6] will determine if other methods perform better than simulated annealing.

- **Using variable instance cost:** An important assumption in this report (see Eq. 1.3) is that the cost of the instance on which a task is being processed is fixed. Amazon Web Services has the concept of Spot Instances [1], where temporarily unused capacity is sold at deeply discounted prices call spot prices which are 70-90% lower than regular prices. Customers specify a maximum price they are willing to pay for an instance and retain the instance as long as the spot price remains below this maximum value. The spot price fluctuates based on supply and demand and can vary a lot within the same hour, sometimes even going above the fixed price for the instance. This setup allows Amazon to earn income on idle capacity and gives customers a much cheaper way to process their tasks as long as they are willing to endure interruptions. Taking advantage of this feature requires the development of *pre-emptive* schedules where tasks can be stopped and re-started later and the prediction of future Spot prices based on the most recent set of prices to determine the maximum bid that should be placed on instance price. The significant reduction of processing costs makes this a feature worth exploring further.

Bibliography

[1] Amazon Web Services. *Amazon EC2 Spot Instances*. URL: http://aws.amazon.com/ec2/purchasing-options/spot-instances/.

[2] Amazon Web Services. *Amazon Elastic Compute Cloud*. URL: https://aws.amazon.com/ec2/.

[3] José M. Bernardo and Adrian F. M. Smith. *Bayesian Theory*. Wiley, 2000. ISBN: 047149464X.

[4] Marco Dorigo, Mauro Birattari, and Thomas Stutzle. "Ant colony optimization". In: *Computational Intelligence Magazine, IEEE* 1.4 (2006), pp. 28–39.

[5] B Efron and R J Tibshirani. *An Introduction to the Bootstrap*. 1st. Chapman and Hall/CRC, 1993, p. 456. ISBN: 0412042312.

[6] Dimitris Fouskakis. "Stochastic Optimization Methods for Cost-Effect Quality Assessment in Health". Ph.D. Thesis. University of Bath, UK, 2001.

[7] M R Garey and D S Johnson. "Computers and Intractability: A Guide to the Theory of NP-Completeness (Series of Books in the Mathematical Sciences)". In: *Computers and Intractability* (1979), p. 340.

[8] C A Glass, C N Potts, and P Shade. "Unrelated parallel machine scheduling using local search". In: *Mathematical and Computer Modelling* 20.2 (1994), pp. 41–52.

[9] F Glover. "Tabu Search - Part I". In: *ORSA Journal on Computing* 1.3 (1989), pp. 190–206.

[10] F. Glover. "Tabu Search - Part II". In: *INFORMS Journal on Computing* 2.1 (1990), pp. 4–32.

[11] R. L. Graham. "Bounds on Multiprocessing Timing Anomalies". In: *SIAM J. Appl. Math.* 17.2 (1969), pp. 416–429.

[12] J.H. Holland. *Adaptation in Natural and Artificial Systems*. 2nd edition. Cambridge, MA: MIT Press, 1992.

[13] Holger Hoos and Thomas Stutzle. *Stochastic Local Search: Foundations and Applications*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2004. ISBN: 1558608729. URL: http://dl.acm.org/citation.cfm?id=983505.

[14] Interoute. *What is Cloud Computing?* 2015. URL: http://www.interoute.com/cloud-article/what-cloud-computing.

[15] S Kirkpatrick, C D Gelatt, and M P Vecchi. "Optimization by simulated annealing." In: *Science (New York, N.Y.)* 220.4598 (1983), pp. 671–680.

[16] Michael L. Pinedo. *Scheduling: Theory, algorithms, and systems*. 4th ed. Springer, 2012, pp. 1–676. ISBN: 1461419867.

[17] RightScale. *State of the Cloud Report*. 2015. URL: http://www.rightscale.com.

[18] Eric W Weisstein. *Bell Number*. {*From MathWorld—A Wolfram Web Resource*}. 2015. URL: http://mathworld.wolfram.com/BellNumber.html.

Appendix A

Code to generate the results in Chapters 2 and 3

This is the code to generate the plots and tables in Chapters 2 and 3

```
rm(list = ls())

# Install devtools and use devtools to install schedulr from
# GitHub to run simulated annealing code. This is done only
# once for each new version of schedulr
# install.packages('devtools')
# devtools::install_github('niranjv/schedulr', ref='develop')

# load schedulr & setup exponential runtimes dataset
library(schedulr)

data(m3xlarge.runtimes.expdist)
data.env = setup.trainingset.runtimes("m3xlarge", m3xlarge.runtimes.expdist)
rt <- get("m3xlarge.runtimes", envir = data.env)
rts <- get("m3xlarge.runtimes.summary", envir = data.env)


# init vars

# assume 3 instance types available e.g., (m3.large, c3.large,
# c4.large) in AWS EC2 with: processing speed = (3.25, 3.5, 4)
# processing cost = (0.14, 0.105, 0.116)
instance.types <- c("m3.large", "c3.large", "c4.large")
instance.speed <- c(3.25, 3.5, 4)
instance.costs <- c(0.14, 0.105, 0.116)
```

```r
#' Create validation dataset
#'
#' Runtime for each tasks can have a different distribution
#' Input array has the name of the distribution of the runtime for each task
#'
#' @param runtimes.dist Array of strings representing the distribution of
#' runtimes. Currently, the distributions must be one of
#' ('unif', 'poisson', 'gamma', 'exp').
#' @return A list containing the details of the distribution of runtimes for
#' each task in the input set of tasks
create.validation.data <- function(runtimes.dist) {

    num.tasks <- NROW(runtimes.dist)
    num.instance.types = NROW(instance.types)

    simulated.runtimes <- matrix(nrow = num.tasks, ncol = num.instance.types)
    colnames(simulated.runtimes) <- paste("runtimes.", instance.types,
        sep = "")

    means <- matrix(nrow = num.tasks, ncol = num.instance.types)
    vars <- matrix(nrow = num.tasks, ncol = num.instance.types)
    dist.params <- list()

    runtimes <- 1 + (1:length(runtimes.dist))/60  # baseline for params
    for (i in 1:length(runtimes.dist)) {

        param.a <- runtimes[i] * instance.speed[1]
        param.b <- runtimes[i] * instance.speed[2]
        param.c <- runtimes[i] * instance.speed[3]

        # Discrete Uniform
        if (runtimes.dist[i] == "unif") {

            means[i, 1] <- (1 + param.a)/2
            means[i, 2] <- (1 + param.b)/2
            means[i, 3] <- (1 + param.c)/2

            simulated.runtimes[i, 1] <- runif(1, 1, param.a)
            simulated.runtimes[i, 2] <- runif(1, 1, param.b)
            simulated.runtimes[i, 3] <- runif(1, 1, param.c)
```

```
    vars[i, 1] <- ((param.a - 1)^2)/12
    vars[i, 2] <- ((param.b - 1)^2)/12
    vars[i, 3] <- ((param.c - 1)^2)/12

    dist.params[[i]] <- list(dist = "unif", params = c(param.a,
        param.b, param.c))
}

# Gamma
if (runtimes.dist[i] == "gamma") {

    means[i, 1] <- param.a^2
    means[i, 2] <- param.b^2
    means[i, 3] <- param.c^2

    simulated.runtimes[i, 1] <- rgamma(1, shape = param.a,
        scale = param.a)
    simulated.runtimes[i, 2] <- rgamma(1, shape = param.b,
        scale = param.b)
    simulated.runtimes[i, 3] <- rgamma(1, shape = param.c,
        scale = param.c)

    vars[i, 1] <- param.a^3
    vars[i, 2] <- param.b^3
    vars[i, 3] <- param.c^3

    dist.params[[i]] <- list(dist = "gamma", params = c(param.a,
        param.b, param.c))
}

# Poisson
if (runtimes.dist[i] == "poisson") {

    param.a <- round(param.a)
    param.b <- round(param.b)
    param.c <- round(param.c)

    means[i, 1] <- param.a
    means[i, 2] <- param.b
    means[i, 3] <- param.c

    simulated.runtimes[i, 1] <- rpois(1, param.a)
```

```
            simulated.runtimes[i, 2] <- rpois(1, param.b)
            simulated.runtimes[i, 3] <- rpois(1, param.c)

            vars[i, 1] <- param.a
            vars[i, 2] <- param.b
            vars[i, 3] <- param.c

            dist.params[[i]] <- list(dist = "poisson", params = c(param.a,
                param.b, param.c))
        }

        # Exponential
        if (runtimes.dist[i] == "exp") {

            means[i, 1] <- 1/param.a
            means[i, 2] <- 1/param.b
            means[i, 3] <- 1/param.c

            simulated.runtimes[i, 1] <- rexp(1, param.a)
            simulated.runtimes[i, 2] <- rexp(1, param.b)
            simulated.runtimes[i, 3] <- rexp(1, param.c)

            vars[i, 1] <- 1/(param.a^2)
            vars[i, 2] <- 1/(param.b^2)
            vars[i, 3] <- 1/(param.c^2)

            dist.params[[i]] <- list(dist = "exp", params = c(param.a,
                param.b, param.c))
        }
    }

    return(list(simulated.runtimes = simulated.runtimes, means = means,
        vars = vars, dist.params = dist.params))

}  # end function - create.validation.data


#' Calculate makespan for deterministic runtimes
#'
#' For deterministic runtimes, load is the sum of runtimes of tasks assigned
#' to an instance and makespan is the max of load across all instances.
#'
```

```
#' @param instance.types Array of instance types under consideration
#' @param instance.costs Array of cost per hour for above instance types
#' @param benefit Benefit of completing all tasks by deadline
#' @param deadline Time by which all costs must be complete
#' @param runtimes Deterministic runtimes for all tasks for all instance types
#' @return A list containing the maximum utility and the instance type and
#' makespan associated with this utility
get.schedule.deterministic.runtimes <- function(instance.types,
    instance.costs, benefit, deadline, runtimes) {

    makespan <- ceiling(apply(runtimes, 2, sum))
    makespan.feasible <- makespan[makespan <= deadline]
    instance.costs.feasible <- instance.costs[makespan <= deadline]
    instance.types.feasible <- instance.types[makespan <= deadline]

    util.feasible <- benefit - (instance.costs.feasible * makespan.feasible)
    max.util.idx <- which.max(util.feasible)
    max.util <- util.feasible[max.util.idx]
    max.util.instance.type <- instance.types.feasible[max.util.idx]
    max.util.makespan <- makespan.feasible[max.util.idx]

    return(list(max.util = max.util, max.util.instance.type = max.util.instance.type,
        max.util.makespan = max.util.makespan))

}  # end function - get.schedule.deterministic.runtimes


#' Calculate schedule for large number of tasks whose runtimes are distributed
#' according to known distributions
#'
#' @param instance.types Array of instance types under consideration
#' @param instance.costs Array of cost per hour for above instance types
#' @param benefit Benefit of completing all tasks by deadline
#' @param deadline Time by which all costs must be complete
#' @param means Array of means of runtime distributions
#' @param vars Array of variances of runtime distributions
#' @param feasible.pctl Percentile of makespan distribution. A schedule is
#' feasible only if the deadline is greater than this percentile of the
#' makespan distribution
#' @return A list containing the maximum utility, instance type and details of
#' the makespan associated with this utility
get.schedule.stochastic.runtimes <- function(instance.types, instance.costs,
```

```
    benefit, deadline, means, vars, feasible.pctl) {

    stopifnot(dim(means) == dim(vars))

    means.sum <- apply(means, 2, sum)
    vars.sum <- apply(vars, 2, sum)
    sds <- sqrt(vars.sum)

    makespan.pctl <- qnorm(feasible.pctl, means.sum, sds)
    makespan.pctl <- ceiling(makespan.pctl)

    feasible.idx <- makespan.pctl <= deadline
    makespan.feasible <- makespan.pctl[feasible.idx]
    instance.cost.feasible <- instance.costs[feasible.idx]
    instance.types.feasible <- instance.types[feasible.idx]

    util.feasible <- benefit - (instance.cost.feasible * makespan.feasible)
    max.util <- max(util.feasible)
    max.util.idx <- which.max(util.feasible)

    max.util.instance.type <- instance.types.feasible[max.util.idx]
    max.util.makespan.feasible.pctl <- makespan.feasible[max.util.idx]

    # assuming we are summing over enough tasks that the makespn
    # dist. is Normal
    max.util.makespan.mean <- means.sum[max.util.idx]
    max.util.makespan.sd <- sds[max.util.idx]
    max.util.makespan.lo <- max.util.makespan.mean - 1.96 * max.util.makespan.sd
    max.util.makespan.hi <- max.util.makespan.mean + 1.96 * max.util.makespan.sd

    return(list(max.util = max.util, max.util.instance.type = max.util.instance.type,
        max.util.makespan.feasible.pctl = max.util.makespan.feasible.pctl,
        max.util.makespan.mean = max.util.makespan.mean, max.util.makespan.lo = max.u
        max.util.makespan.hi = max.util.makespan.hi))

}  # end function - get.schedule.stochastic.runtimes


#' Get validation results when makespan distributions are Normally distributed
#'
#' @param instance.types Array of instance types under consideration
#' @param instance.costs Array of cost per hour for above instance types
```

```
#' @param feasible.pctl Percentile of makespan distribution. A schedule is
#' feasible only if the deadline is greater than this percentile of the
#' makespan distribution
#' @param num.tasks Number of input tasks
#' @param num.trials Number of simulated data sets to process
#' @return Nothing. This function generates a plot as a side-effect
validate.stochastic.runtimes <- function(instance.types, instance.costs,
    feasible.pctl = 0.95, num.tasks, num.trials = 1000) {

    validation.results <- data.frame()

    for (j in 1:num.trials) {

        # cat('Processing iteration', j, '\n')

        dist <- c("unif", "poisson", "gamma", "exp")
        runtimes.dist <- sample(dist, num.tasks, replace = T)
        validation.data <- create.validation.data(runtimes.dist)

        # a realistic benefit & deadline or no schedule will be found
        deadline = round(4 * sum(validation.data$simulated.runtimes[,
            1]))
        benefit = deadline

        result.actual = get.schedule.deterministic.runtimes(instance.types,
            instance.costs, benefit, deadline, validation.data$simulated.runtimes)

        result.predicted = get.schedule.stochastic.runtimes(instance.types,
            instance.costs, benefit, deadline, validation.data$means,
            validation.data$vars, feasible.pctl)

        validation.results[j, 1] <- result.actual$max.util
        validation.results[j, 2] <- result.actual$max.util.instance.type
        validation.results[j, 3] <- result.actual$max.util.makespan

        validation.results[j, 4] <- result.predicted$max.util
        validation.results[j, 5] <- result.predicted$max.util.instance.type
        validation.results[j, 6] <- result.predicted$max.util.makespan.mean
        validation.results[j, 7] <- result.predicted$max.util.makespan.lo
        validation.results[j, 8] <- result.predicted$max.util.makespan.hi
    }
```

```
    validation.results <- validation.results[order(validation.results[,
        7]), ]
    validation.results <- cbind(1:NROW(validation.results), validation.results)
    colnames(validation.results) <- c("index", "actual.util", "actual.inst",
        "actual.makespan", "pred.util", "pred.inst", "pred.makespan.mean",
        "pred.makespan.lo", "pred.makespan.hi")

    plot.validation.results(validation.results, num.tasks, num.trials)

}


get.runtime.dist.bootstrap <- function(num.instance.types, dist.params) {

    num.tasks <- length(dist.params)

    runtime.dist <- matrix(nrow = num.tasks, ncol = num.instance.types)

    for (j in 1:num.tasks) {
        dist.type <- dist.params[[j]]$dist
        params <- dist.params[[j]]$params
        stopifnot(num.instance.types <= length(params))

        # Discrete Uniform
        if (dist.type == "unif") {
            runtime.dist[j, ] <- runif(num.instance.types, min = 1,
                max = params)
        }

        # Gamma
        if (dist.type == "gamma") {
            runtime.dist[j, ] <- rgamma(num.instance.types, shape = params,
                scale = params)
        }

        # Poisson
        if (dist.type == "poisson") {
            runtime.dist[j, ] <- rpois(num.instance.types, lambda = params)
        }

        # Exponential
        if (dist.type == "exp") {
```

```
            runtime.dist[j, ] <- rexp(num.instance.types, rate = params)
        }

    }  # loop over dist. for all tasks


    return(runtime.dist)

}  # end function - get.runtime.dist.bootstrap


#' Get runtime dist via bootstrap resampling, then get schedule
#'
#' @param instance.types Array of instance types under consideration
#' @param instance.costs Array of cost per hour for above instance types
#' @param num.tasks Number of input tasks
#' @param num.trials Number of simulated data sets to process
#' @param num.bootstrap.reps Number of bootstrap samples to use while generating
#' runtime distribution
#' @param feasible.pctl Threshold to use to detemine makespan for runtime dist.
#' @return Nothing. This function generates a plot as a side-effect
get.schedule.stochastic.runtimes.bootstrap <- function(instance.types,
    instance.costs, benefit, deadline, dist.params, num.bootstrap.reps,
    feasible.pctl) {

    num.instance.types <- length(instance.types)

    makespan.bootstrap.dist <- matrix(nrow = num.bootstrap.reps,
        ncol = num.instance.types)

    for (i in 1:num.bootstrap.reps) {

        if (1%%10 == 0) {
            cat("Processing trial", i, "\n")
        }

        runtime.dist <- get.runtime.dist.bootstrap(num.instance.types,
            dist.params)
        makespan.bootstrap.dist[i, ] <- apply(runtime.dist, 2,
            sum)
    }
```

```
    makespan.pct <- apply(makespan.bootstrap.dist, 2, quantile,
        prob = feasible.pctl)
    makespan.pct <- ceiling(makespan.pct)

    makespan.feasible <- makespan.pct[makespan.pct <= deadline]
    instance.cost.feasible <- instance.costs[makespan.pct <= deadline]
    instance.types.feasible <- instance.types[makespan.pct <= deadline]

    util.feasible <- benefit - (instance.cost.feasible * makespan.feasible)
    max.util.idx <- which.max(util.feasible)


    return(list(makespan.feasible = makespan.feasible, util.feasible = util.feasible,
        max.util = util.feasible[max.util.idx], max.util.dist = makespan.bootstrap.di
            max.util.idx], max.util.instance.type = instance.types.feasible[max.util.
        max.util.makespan.feasible.pctl = makespan.feasible[max.util.idx],
        max.util.makespan.mean = mean(makespan.bootstrap.dist[,
            max.util.idx]), max.util.makespan.var = var(makespan.bootstrap.dist[,
            max.util.idx]), max.util.makespan.lo = quantile(makespan.bootstrap.dist[,
            max.util.idx], prob = 0.025), max.util.makespan.hi = quantile(makespan.bo
            max.util.idx], prob = 0.975)))

} # end function - get.schedule.stochastic.runtimes.bootstrap


#' Get validation results when runtime distributions are obtained via bootstrap
#' sampling
#'
#' @param instance.types Array of instance types under consideration
#' @param instance.costs Array of cost per hour for above instance types
#' @param num.tasks Number of input tasks
#' @param num.trials Number of simulated data sets to process
#' @param num.bootstrap.reps Number of bootstrap samples to use while generating
#' runtime distribution
#' @param feasible.pctl Threshold to use to detemine makespan for runtime dist.
#' @return Nothing. This function generates a plot as a side-effect
validate.stochastic.runtimes.bootstrap <- function(instance.types,
    instance.costs, num.tasks, num.trials = 1000, num.bootstrap.reps = 1000,
    feasible.pctl = 0.95) {

    validation.results <- data.frame()
```

```
for (j in 1:num.trials) {

    if (j%%100 == 0) {
        cat("Processing trial", j, "\n")
    }

    dist.list <- c("unif", "poisson", "gamma", "exp")
    runtimes.dist <- sample(dist.list, num.tasks, replace = T)
    runtimes.dist <- rep("poisson", num.tasks)
    validation.data <- create.validation.data(runtimes.dist)

    # a realistic benefit & deadline or no schedule will be found
    deadline = round(4 * sum(validation.data$simulated.runtimes[,
        1]))
    benefit = deadline

    # this is the makespan if the runtimes were deterministic
    result.actual = get.schedule.deterministic.runtimes(instance.types,
        instance.costs, benefit, deadline, validation.data$simulated.runtimes)

    result.predicted = get.schedule.stochastic.runtimes.bootstrap(instance.types,
        instance.costs, benefit, deadline, validation.data$dist.params,
        num.bootstrap.reps, feasible.pctl)

    validation.results[j, 1] <- result.actual$max.util
    validation.results[j, 2] <- result.actual$max.util.instance.type
    validation.results[j, 3] <- result.actual$max.util.makespan

    validation.results[j, 4] <- result.predicted$max.util
    validation.results[j, 5] <- result.predicted$max.util.instance.type
    validation.results[j, 6] <- result.predicted$max.util.makespan.mean
    validation.results[j, 7] <- result.predicted$max.util.makespan.lo
    validation.results[j, 8] <- result.predicted$max.util.makespan.hi
}

validation.results <- validation.results[order(validation.results[,
    7]), ]
validation.results <- cbind(1:NROW(validation.results), validation.results)
colnames(validation.results) <- c("index", "actual.util", "actual.inst",
    "actual.makespan", "pred.util", "pred.inst", "pred.makespan.mean",
    "pred.makespan.lo", "pred.makespan.hi")
```

```
    plot.validation.results(validation.results, num.tasks, num.trials)

}  # end function - validate.stochastic.runtimes.bootstrap


#' Plot validation results
#'
#' Plot actual runtimes with 95% CI for predicted runtimes
#'
#' @param validation.results Matrix containing the columns (
#'  Actual maximum utility,
#'  Actual instance with maximum utility,
#'  Actual makespan with maximum utility,
#'  Predicted maximum utility,
#'  Predicted instance type with maximum utility,
#'  Mean of predicted makespan with maximum utility,
#'  Lower bound of 95% CI of predicted makespan with maximum utility,
#'  Higher bound of 95% CI of predicted makespan with maximum utility,
#' )
#' @param img.title Title to use in image
#' @param img.filename Path to image file
#' @return Nothing. This function is called for the side-effect of generating
#' a plot in a file.
plot.validation.results <- function(validation.results, num.tasks,
    num.trials) {

    # We can compare runtimes only when both the actual and
    # predicted schedules use the same instance type
    validation.results.matched <- subset(validation.results, validation.results[,
        3] == validation.results[, 6])

    outliers <- subset(validation.results.matched, validation.results.matched[,
        4] < validation.results.matched[, 8] | validation.results.matched[,
        4] > validation.results.matched[, 9])

    outliers.pct <- round(100 * NROW(outliers)/NROW(validation.results.matched),
        2)

    img.title <- paste("95% CI for makespan\n", sep = "")
    img.title <- paste(img.title, NROW(validation.results), " trials; ",
        num.tasks, " tasks/trial\n", sep = "")
    img.title <- paste(img.title, outliers.pct, "% outliers in the ",
```

```
        round(100 * NROW(validation.results.matched)/NROW(validation.results),
            2), "% of trials with matched instance types", sep = "")

    img.filename <- paste("validate-stochastic-runtimes-", num.trials,
        "-trials-", num.tasks, "-tasks.eps", sep = "")

    y.lim <- round(range(validation.results[, c(4, 8, 9)]))

    img.dir <- "content/figures"
    if (!file.exists(img.dir)) {
        dir.create(img.dir)
    }

    img.filepath <- file.path(img.dir, img.filename)

    postscript(img.filepath, height = 7, width = 7, onefile = FALSE,
        horizontal = FALSE)
    plot(validation.results[, 1], validation.results[, 4], pch = 16,
        cex = 0.5, main = img.title, ylim = y.lim, xlab = "Trial #",
        ylab = "Makespan (hr)")

    lines(validation.results[, 1], validation.results[, 8], col = "red",
        lty = "dotted", lwd = 2)

    lines(validation.results[, 1], validation.results[, 9], col = "red",
        lty = "dotted", lwd = 2)

    legend("bottomright", legend = "95% CI for predicted makespan",
        col = "red", lty = "dotted", lwd = 2)
    dev.off()

    cat("Created plot:", img.filepath, "\n")

}  # end function - plot.validation.results

# ==========

# Validate schedule - 1 instance; 100 tasks; makespan
# distribution approximated by Normal distribution
start.time <- proc.time()
validate.stochastic.runtimes(instance.types, instance.costs, num.tasks = 100)
cat("Time taken: ", round((proc.time() - start.time)[3]/60, 2),
    " mins")
```

```
# Validate schedule - 1 instance; 250 tasks; makespan
# distribution approximated by Normal distribution
start.time <- proc.time()
validate.stochastic.runtimes(instance.types, instance.costs, num.tasks = 250)
cat("Time taken: ", round((proc.time() - start.time)[3]/60, 2),
    " mins")

# Validate schedule - 1 instance; 500 tasks; makespan
# distribution approximated by Normal distribution
start.time <- proc.time()
validate.stochastic.runtimes(instance.types, instance.costs, num.tasks = 500)
cat("Time taken: ", round((proc.time() - start.time)[3]/60, 2),
    " mins")

# Validate schedule - 1 instance; 1000 tasks; makespan
# distribution approximated by Normal distribution
start.time <- proc.time()
validate.stochastic.runtimes(instance.types, instance.costs, num.tasks = 1000)
cat("Time taken: ", round((proc.time() - start.time)[3]/60, 2),
    " mins")

# ==========

# Validate schedule - 1 instance; 10 tasks; makespan
# distribution approximated by bootstrap re-sampling
start.time <- proc.time()
validate.stochastic.runtimes.bootstrap(instance.types, instance.costs,
    num.tasks = 10)
cat("Time taken: ", round((proc.time() - start.time)[3]/60, 2),
    " mins")

# Validate schedule - 1 instance; 25 tasks; makespan
# distribution approximated by bootstrap re-sampling
start.time <- proc.time()
validate.stochastic.runtimes.bootstrap(instance.types, instance.costs,
    num.tasks = 25)
cat("Time taken: ", round((proc.time() - start.time)[3]/60, 2),
    " mins")

# Validate schedule - 1 instance; 50 tasks; makespan
# distribution approximated by bootstrap re-sampling
start.time <- proc.time()
validate.stochastic.runtimes.bootstrap(instance.types, instance.costs,
```

```
    num.tasks = 50)
cat("Time taken: ", round((proc.time() - start.time)[3]/60, 2),
    " mins")

# Validate schedule - 1 instance; 75 tasks; makespan
# distribution approximated by bootstrap re-sampling
start.time <- proc.time()
validate.stochastic.runtimes.bootstrap(instance.types, instance.costs,
    num.tasks = 75)
cat("Time taken: ", round((proc.time() - start.time)[3]/60, 2),
    " mins")
```

Appendix B

Code for *schedulr* R package

This is the code for the schedulr R package

```
## @knitr all

# Functions for Simulated annealing


data.env <- new.env()

# If an instance has more than bootstrap.threshold tasks, use Normal approx.
# to get runtime dist., instead of generating bootstrap samples
bootstrap.threshold <- 50
num.bootstrap.reps <- 1000

instance.types <- c('m3.xlarge')
instance.speed <- c(3.25)
instance.costs <- c(0.28)

# -----
# Internal functions for validating input
# -----


#' Validate that the input value is a positive integer (test single number,
#' not array)
#'
#' @param val The value to validate
#' @examples
#' check.if.positive.integer()
#' check.if.positive.integer(c())
```

```
#' check.if.positive.integer('')
#' check.if.positive.integer(5)
#' check.if.positive.integer(0)
#' check.if.positive.integer(-10)
#' check.if.positive.integer(3.14)
#' check.if.positive.integer(1:2)
#' check.if.positive.integer('a')
.check.if.positive.integer <- function (value) {

  .check.if.nonnegative.integer(value)
  value > 0 || stop("Invalid argument: Value must be > 0")

} # end function - .check.if.positive.integer


#' Validate that the input value is a non-negative integer (test single number,
#' not array)
#'
#' @param val The value to validate
#' @examples
#' check.if.nonnegative.integer()
#' check.if.nonnegative.integer(c())
#' check.if.nonnegative.integer('')
#' check.if.nonnegative.integer(5)
#' check.if.nonnegative.integer(0)
#' check.if.nonnegative.integer(-10)
#' check.if.nonnegative.integer(3.14)
#' check.if.nonnegative.integer(1:2)
#' check.if.nonnegative.integer('a')
.check.if.nonnegative.integer <- function (value) {

  !missing(value) || stop("Missing required argument: Must specify a value")
  length(value) == 1 || stop("Invalid argument length:
    Must specify a single number")
  (is.numeric(value) && value == floor(value)) || stop('Non-integer argument:
    value')
  value >= 0 || stop("Invalid argument: Value must be >= 0")

} # end function - .check.if.nonnegative.integer


#' Verify that the input value is a positive real (test arrays)
```

```r
#'
#' @param value Array of values to validate
#' @examples
#' .check.if.positive.real()
#' .check.if.positive.real(c())
#' .check.if.positive.real('')
#' .check.if.positive.real(0)
#' .check.if.positive.real(1)
#' .check.if.positive.real(3.14)
#' .check.if.positive.real(-5)
#' .check.if.positive.real(c(1.2, 3.4))
#' .check.if.positive.real('a')
.check.if.positive.real <- function (value) {

  .check.if.nonnegative.real(value)
  all(value > 0) || stop('Invalid argument: Value must be > 0')

} # end function - .check.if.positive.real


#' Verify that the input value is a non-negative real (test arrays)
#'
#' @param value Array of values to validate
#' @examples
#' .check.if.nonnegative.real()
#' .check.if.nonnegative.real(c())
#' .check.if.nonnegative.real('')
#' .check.if.nonnegative.real(0)
#' .check.if.nonnegative.real(1)
#' .check.if.nonnegative.real(c(1.2, 3.4))
#' .check.if.nonnegative.real(3.14)
#' .check.if.nonnegative.real('a')
.check.if.nonnegative.real <- function (value) {

  !missing(value) || stop('Missing required argument: Must specify a value')
  length(value) > 0 || stop('Invalid argument length: Must specify a value')
  is.numeric(value) || stop('Non-numeric argument:
    Must specify a valid +ve real number')
  all(value >= 0) || stop('Invalid argument: Value must be >= 0')

} # end function - .check.if.nonnegative.real
```

```
#' Verify that schedule is valid
#'
#' @param schedule Array of task sizes
#' @examples
#' a <- get.initial.schedule(2, 3)
#' .validate.schedule(a)
#' .validate.schedule(b<-NULL)
.validate.schedule <- function (schedule) {

  !missing(schedule) || stop("Missing required argument: schedule")
  is.list(schedule) || stop("Invalid argument type:
    schedule must be a list")
  length(schedule) != 0 || stop("Invalid argument length:
    schedule must contain at least 1 instance")
  is.numeric(unlist(schedule)) || stop("Non-numeric argument:
    tasks sizes must be valid numbers")
  sum(unlist(schedule) <= 0) == 0 || stop("Invalid argument:
    tasks sizes must be > 0")

} # end function - .validate.schedule



#' Verify that schedule attributes are valid
#'
#' @param schedule Array of task sizes
#' @examples
#' a <- get.initial.schedule(2, c(10))
#' .validate.schedule.attributes(a)
#' attr(a, 'score') <- 0
#' attr(a, 'runtime95pct') <- 0
#' attr(a, 'runtime99pct') <- 0
#' .validate.schedule.attributes(a)
.validate.schedule.attributes <- function (schedule) {

  is.numeric(attr(schedule, 'score')) || stop("Invalid argument:
    schedule score must be a valid number")
  attr(schedule, 'score') >= 0 || stop("Invalid argument:
    schedule score must be >= 0")

  is.numeric(attr(schedule, 'deadline')) || stop("Invalid argument:
```

```
    schedule deadline must be a valid number")
  attr(schedule, 'deadline') > 0 || stop("Invalid argument:
    deadline must be > 0")

  is.numeric(attr(schedule, 'runtime95pct')) || stop("Invalid argument:
    schedule runtime95pct must be a valid number")
  attr(schedule, 'runtime95pct') >= 0 || stop("Invalid argument:
    schedule runtime95pct must be >= 0")

  is.numeric(attr(schedule, 'runtime99pct')) || stop("Invalid argument:
    schedule runtime99pct must be a valid number")
  attr(schedule, 'runtime99pct') >= 0 || stop("Invalid argument:
    schedule runtime99pct must be >= 0")

} # end function - .validate.schedule



#' Verify that the schedule has the minimum number of tasks required
#'
#' @param schedule List mapping tasks to instances
#' @param min.num.tasks Minimum number of tasks in schedule
#' @examples
#' a <- get.initial.schedule(2, c(10))
#' .validate.num.tasks.in.schedule(a, 2)
#' .validate.num.tasks.in.schedule(a, 5)
.validate.num.tasks.in.schedule <- function (schedule, num.tasks.required) {

  num.tasks.available <- length(unlist(schedule))
  if (num.tasks.available >= num.tasks.required) {
    return (TRUE)
  } else {
    return (FALSE)
  } # end if - move more tasks than available?

} # end function .validate.num.tasks.in.schedule


#' Verify that runtimes are valid values
#'
#' @param runtimes Matrix of runtime of past runs for the given instance type
#' Each row in the matrix represents a single training sample and has 2 columns.
```

```
#' The size column is the size of task that was processed.
#' The runtime_sec column is the time taken to process the task in seconds.
#' @examples
#' r <- matrix(c(1,1), nrow=1, ncol=2)
#' .validate.runtimes.summary(r)
.validate.runtimes <- function (runtimes) {

  !missing(runtimes) || stop("Missing required argument:
    Must specify a numeric matrix with 2 columns")
  is.matrix(runtimes) || stop("Invalid argument type:
    Must specify a numeric matrix with 2 columns")
  NCOL(runtimes) == 2 || stop("Invalid argument dimensions:
    Must specify a numeric matrix with 2 columns")
  NROW(runtimes) > 0 || stop("Invalid argument dimensions:
    Must specify a numeric matrix with 2 columns and at least 1 row")
  is.numeric(runtimes) || stop ("Invalid argument:
    Must specify a numeric matrix with 2 columns")
  all(runtimes[,1] > 0) || stop("Invalid argument:
    1st column (size) must have positive values")
  all(runtimes[,2] >= 0) || stop("Invalid argument:
    2nd column (runtime) must have positive values")

} # end function - .validate.runtimes



#' Verify that runtime summaries are valid values
#'
#' @param runtimes.summary Numeric matrix containing mean
#' and variance of runtimes for each size
#' @examples
#' rs <- matrix(c(1,1,1), nrow=1, ncol=3)
#' .validate.runtimes.summary(rs)
.validate.runtimes.summary <- function (runtimes.summary) {

  !missing(runtimes.summary) || stop("Missing required argument:
    Must specify a numeric matrix with 2 columns")
  is.matrix(runtimes.summary) || stop("Invalid argument type:
    Must specify a numeric matrix with 2 columns")
  NCOL(runtimes.summary) == 3 || stop("Invalid argument dimensions:
    Must specify a numeric matrix with 3 columns")
  NROW(runtimes.summary) > 0 || stop("Invalid argument dimensions:
```

```
    Must specify a numeric matrix with 2 columns and at least 1 row")
  is.numeric(runtimes.summary) || stop ("Invalid argument:
    Must specify a numeric matrix with 2 columns")
  all(runtimes.summary[,1] > 0) || stop("Invalid argument:
    1st column (size) must have positive values")
  all(runtimes.summary[,2] > 0) || stop("Invalid argument:
    2nd column (runtime) must have positive values")
  all(runtimes.summary[,3] >= 0) || stop("Invalid argument:
  3rd column (var(runtimes)) cannot have negative values")

} # end function - .validate.runtimes.summary



.validate.instance.type <- function (instance.type) {

  !missing(instance.type) || stop("Missing required argument:
    Must specify instance.type")
  length(instance.type) != 0 || stop("Invalid argument length:
    instance.type must be a string")
  nchar(instance.type) > 0 || stop("Invalid argument length:
    instance.type must be a string")
  is.character(instance.type) || stop ("Invalid argument type:
    instance.type must be a string")
  NROW(instance.type) == 1 || stop ("Invalid argument length:
    instance.type must be a string, not a vector of strings")

} # end function - .validate.runtimes


# -----
# Other internal functions
# -----


#' Get runtimes for instance type
#'
#' @inheritParams setup.trainingset.runtimes
#' @param summary Return only summary of runtimes.
#' @return
#' If summary=F, return value is a matrix of runtimes for the given
#' instance type.
```

```
#' Each row in the matrix represents a single trial and has 2 columns.
#' The 1st column is the size of task that was processed and
#' the 2nd column is the runtime for this size.
#' If summary=T, return value is a matrix of summary of runtimes for the given
#' instance type. Each row in the matrix represents a single size and has
#' 3 columns.
#' The 1st column is the size of task that was processed,
#' the 2nd column is the mean runtime for this size and
#' the 3rd column is the variance of the runtimes for this size
#' @examples
#' .get.trainingset.runtimes('m3xlarge')
.get.trainingset.runtimes <- function (instance.type, summary=F) {

  if (summary) {
    varname <- paste(instance.type, '.runtimes.summary', sep='')
  } else {
    varname <- paste(instance.type, '.runtimes', sep='')
  } # end if - get summary?

  exists(varname, envir=data.env) ||
    stop("Runtimes for ", instance.type, " not setup correctly")
  var <- get(varname, envir=data.env) # get var from internal env (data.env)
  return (var)

} # end function - .get.trainingset.runtimes



#' Get initial schedule of tasks to instances in a cluster
#'
#' Tasks are randomly assigned to instances
#'
#' @inheritParams get.initial.schedule
#' @return List containing a mapping of tasks to instances in cluster.
#' The list index represents the id of an instance in the cluster while
#' the associated list member represents the task assigned to that instance
#' @examples
#' schedule <- get.initial.schedule.random(4, 1:30)
.get.initial.schedule.random <- function (cluster.size, task.sizes) {

  schedule <- vector('list', cluster.size)
  num.tasks <- length(task.sizes)
```

```
  idx.shuffle <- sample(num.tasks, replace=F)
  shuffled.task.sizes <- task.sizes[idx.shuffle]

  for (i in 1:num.tasks) {

    # get random instance
    inst <- sample(length(schedule), 1)
      schedule[[inst]] <- c(schedule[[inst]], shuffled.task.sizes[i])

  } # end for - loop over all tasks in order

  return (schedule)

} # end function - get.initial.schedule.random



#' Get initial schedule of tasks to instances in a cluster
#'
#' Tasks are assigned to instances in decreasing order of expected processing
#' time (i.e., Longest Expected Processing Time First rule)
#'
#' @inheritParams get.initial.schedule
#' @return List containing a mapping of tasks to instances in cluster.
#' The list index represents the id of an instance in the cluster while the
#' associated list member represents the task assigned to that instance
#' @examples
#' rs <- matrix(nrow=2, ncol=3)
#' rs[1,1] <- 10; rs[1,2] <- 23.5; rs[1,3] <- 2.5
#' rs[2,1] <- 20; rs[2,2] <- 33.5; rs[2,3] <- 3.5
#' schedule <- get.initial.schedule.leptf(2, rep(c(1,2), 3), rs)
.get.initial.schedule.leptf <- function (cluster.size, task.sizes,
  runtimes.summary) {

  schedule <- vector('list', cluster.size)
  # to keep track of total runtimes in each instance
  total.runtimes <- array(0, dim=cluster.size)
  num.tasks <- length(task.sizes)

  means <- sapply(task.sizes, function (x) {
    idx <- which(runtimes.summary[,1] == x); return(runtimes.summary[idx,2])
  })
```

```
  size.means <- cbind(task.sizes, means)
  size.means <- size.means[order(size.means[,2], decreasing=TRUE), ]
  if (class(size.means) == 'numeric') size.means <- as.matrix(t(size.means))
  colnames(size.means) <- NULL
  rownames(size.means) <- NULL

  for (i in 1:num.tasks) {

      instance.with.smallest.total.runtime <- which.min(total.runtimes)
      # if multiple elements in list have the lowest value,
    # which.min returns the first. For our purposes, it doesn't matter which of
    # the instances with the lowest total is used next.

      schedule[[instance.with.smallest.total.runtime]] <-
    c(schedule[[instance.with.smallest.total.runtime]], size.means[i,1])
    total.runtimes[instance.with.smallest.total.runtime] <-
    total.runtimes[instance.with.smallest.total.runtime] + size.means[i,2]

  } # end for - loop over all tasks in order

  return (schedule)

} # end function - get.initial.schedule.leptf



#' Get list of instances that have the minimum number of tasks required
.get.admissable.instances <- function (schedule, num.tasks.per.instance,
                                       num.instances.to.use) {

  num.tasks.in.instances <- lapply(schedule, length)
  admissable.instances <-
    which(num.tasks.in.instances >= num.tasks.per.instance)
  return (admissable.instances)
} # end function - get.admissable.instances



#' Get number of instances depending on whether to exchange tasks or move tasks
.get.num.instances <- function (exchange) {
  num.instances <- 1
  if (exchange) num.instances <- 2
```

```
  return (num.instances)

} # end function - .get.num.instances



#' Get temperature for current iteration
#'
#' Temperature decreases linearly with each iteration
#'
#' @inheritParams get.temperature
#' @return Value of temperture for the current iteration (integer)
#' @examples
#' temp <- .get.temperature.linear.decrease(25, 100, 7)
.get.temperature.linear.decrease <- function (max.temp, max.iter, cur.iter) {

  # cur.iter is guaranteed to be at most 1 less than max.iter
  # so cur.temp will always be > 0
  cur.temp <- (max.iter-cur.iter)*(max.temp/max.iter)
  return (cur.temp)

} # end function - get.temperature.linear.decrease



#' Get bootstrap sample for a task in the input job
#'
#' @param input.size Task size for which samples are required (integer)
#' @param num.samples Number of samples required  (integer)
#' @param runtimes Matrix containing size & runtime info for training set sample
#' @return Matrix containing required number of samples for the given size
.bootstrap.get.task.sample <- function (input.size, num.samples, runtimes) {

  varname <- paste('runtimes.', input.size, sep='')
  runtimes.cur.size <- get(varname, envir=data.env)
  num.rows <- NROW(runtimes.cur.size)

  num.rows > 0 || stop('Cannot find any samples for size=', input.size,
    ' in training set. Ensure that training set has samples for this task size')

  idx <- sample(1:num.rows, num.samples, replace=T)
```

```
  s <- runtimes.cur.size[idx,]

  # transpose data frames due to the way they are 'flattened' in unlist
  if (NROW(s) > 1) s <- t(s)

  return (s)
} # end function - .bootstrap.get.task.sample


#' Get bootstrapped samples for all sizes in the input job
.bootstrap.get.job.sample <- function (size.re
ps.table, runtimes) {

  # FORMAT of size.reps.table (generated via aggregate())
  # > size.reps.table
      # Group.1 x
  # 1       10 1
  # 2       90 1
  # 3      200 1
  # 4      850 1
  # 5     2100 1

  samples.list <- apply(size.reps.table, 1, function (x) {
    .bootstrap.get.task.sample(x[1], x[2], runtimes)
  })
  samples.matrix <- matrix(unlist(samples.list), ncol=2, byrow=TRUE)

  return (samples.matrix)

} # end function - .bootstrap.get.job.sample


.bootstrap.get.job.runtime <- function (size.reps.table, runtimes) {

  samples.matrix <- .bootstrap.get.job.sample(size.reps.table, runtimes)
  s <- sum(samples.matrix[,2])
  return (s)

} # end function - .bootstrap.get.job.runtime


#' Get distribution of job runtime via bootstrap re-sampling
```

```
#'
#' @param size.reps.table Data frame with 2 columns; typically obtained as the
#' output from the aggregate() function.
#' 1st column is the task size
#' 2nd column is the number of tasks with this size
#' @param num.bootstrap.reps Number of bootstrap replicates in distribution
#' @param runtimes Matrix containing size & runtime info for training set sample
#' @examples
#' data(m3xlarge.runtimes.expdist)
#' setup.trainingset.runtimes('m3xlarge', m3xlarge.runtimes.expdist)
#' job <- c(1,60,100)
#' srt <- aggregate(job, by=list(job), length)
#' dist <- .bootstrap.get.job.runtime.dist(srt, 500, m3xlarge.runtimes.expdist)
.bootstrap.get.job.runtime.dist <-
  function (size.reps.table, num.bootstrap.reps, runtimes) {

  job.runtime.dist <- array(dim=num.bootstrap.reps)
  for(i in 1:num.bootstrap.reps) {
      r <- .bootstrap.get.job.runtime(size.reps.table, runtimes)
    job.runtime.dist[i] <- r
  } # end for - perform required number of iterations

  return (job.runtime.dist)

} # end function - .bootstrap.get.job.runtime.dist



# -----
# Exported functions
# -----


#' Setup runtimes for given instance type
#'
#' All instances in a cluster are assumed to be of the same type
#'
#' @param instance.type Instance type of cluster (string).
#' All instances in the cluster are assumed to be of the same type
#' @param runtimes Matrix of runtimes for the given instance type
#' Each row in the matrix represents a single training sample and has 2 columns.
#' The size column is the size of task that was processed.
```

```
#' The runtime_sec column is the time taken to process the task in seconds.
#' @return The environment in which the varibles were set up
#' @export
#' @examples
#' runtimes <- cbind(rep(c(1,2), each=5), c(rpois(5,5), rpois(5,10)))
#' setup.trainingset.runtimes('m3xlarge', runtimes)
setup.trainingset.runtimes <- function (instance.type, runtimes) {

  # Validate args
  .validate.instance.type(instance.type)
  .validate.runtimes(runtimes)

  # Save runtimes of individual trials to use in bootstrap sampling
  varname <- paste(instance.type, '.runtimes', sep='')
  # create new var in internal env (data.env)
  assign(varname, runtimes, envir=data.env)


  # save runtime summary
  m <- aggregate(runtimes[, 2], by=list(runtimes[, 1]), mean)
  v <- aggregate(runtimes[, 2], by=list(runtimes[, 1]), var)
  mv <- cbind(m[, 1], m[, 2], v[, 2])
  colnames(mv) <- c('size', 'mean', 'var')

  varname <- paste(instance.type, '.runtimes.summary', sep='')
  # create new var in internal env (data.env)
  assign(varname, mv, envir=data.env)


  # save runtimes for each size in a separate var
  uniq.sizes <- unique(runtimes[,1])
  for (s in uniq.sizes) {
    varname <- paste('runtimes.', s, sep='')
    ss <- subset(runtimes, runtimes[,1]==s)
    assign(varname, ss, envir=data.env)
  } # end for - loop over all sizes

  return(data.env)

} # end function - setup.trainingset.runtimes
```

```r
#' Get initial schedule of jobs to instances in a cluster
#'
#' @param cluster.size Number of instances in the cluster (+ve integer)
#' @param task.sizes Array of task sizes (+ve reals)
#' @param runtimes.summary Numeric matrix containing mean and variance of
#' runtimes for each size. Must be supplied when method='leptf'
#' @param method Method to use to assign tasks to instances.
#' Must be one of ('random', 'leptf').
#' @return List containing a mapping of tasks to instances in cluster.
#' The list index represents the id of an instance in the cluster while
#' the associated list member represents the task assigned to that instance
#' @export
#' @examples
#' a <- get.initial.schedule(3, 1:30)
#' rs <- matrix(nrow=2, ncol=3)
#' rs[1,1] <- 10; rs[1,2] <- 23.5; rs[1,3] <- 2.5
#' rs[2,1] <- 20; rs[2,2] <- 33.5; rs[2,3] <- 3.5
#' a <- get.initial.schedule(3, c(rep(10, 3), rep(20, 3)), rs, method='leptf')
get.initial.schedule <-
  function (cluster.size, task.sizes, runtimes.summary, method='random') {

  # Validate args
  .check.if.positive.integer(cluster.size)
  .check.if.positive.real(task.sizes)

  if (method=='random') {
    schedule <- .get.initial.schedule.random(cluster.size, task.sizes)
  } else if (method=='leptf') {
    .validate.runtimes.summary(runtimes.summary)
    schedule <- .get.initial.schedule.leptf(
      cluster.size, task.sizes, runtimes.summary
    )
  } else {
    stop('Invalid argument: ', method, ' is not a valid value for method')
  } # end if - method=random?

  return (schedule)

} # end function - get.initial.schedule
```

```
#' Generate a neighbor to an schedule
#'
#' The input schedule is modified in one of several different ways, including
#' \itemize{
#'  \item Move a task from 1 instance to another
#'  \item Exchange a task with another instance
#'  \item Move 2 tasks from 1 instance to another
#'  \item Exchange 2 tasks with another instance
#'  \item Move 2 tasks from an instance to 2 other instance
#'  \item Exchange 2 tasks with 2 other instances
#'  \item and so on...
#' }
#' Only the first 2 methods are currently implemented with an equal probability
#' of selecting either method.
#'
#' @param schedule A list representing a mapping of tasks to instances in a
#' cluster
#' @return A list representing the modified schedule of tasks to instances in
#' the cluster
#' @export
#' @examples
#' schedule <- get.initial.schedule(3, 1:30)
#' proposed.schedule <- get.neighbor(schedule)
get.neighbor <- function (schedule) {

  # Validate args
  .validate.schedule(schedule)

  # Cannot get neighbors if cluster has < 2 instances
  num.instances.in.schedule <- length(schedule)
  if (num.instances.in.schedule < 2) { return (schedule) }


  ex <- sample(c(TRUE, FALSE), 1)

  num.tasks.in.instances <- sapply(schedule, length)
  num.tasks.in.instances <- round(num.tasks.in.instances/3)
  num.tasks <- sample(max(num.tasks.in.instances), 1)

  if (ex) { cat('Exchange', num.tasks, 'tasks \n\n') }
  else { cat('Move', num.tasks, 'tasks \n\n') }
```

```
  neighbor <- move.tasks(schedule, num.tasks, exchange=ex)

  return (neighbor)

} # end function - get.neighbor



#' Generate neighbor by moving 1 task
#'
#' Randomly select 2 instances in the cluster. Randomly select a task from one
#' of the instances and move it to the other instance. Simple random sampling
#' without replacement is used in both sampling stages.
#'
#' @param schedule A list representing the schedule for which a neighbor is
#' desired
#' @param num.tasks Integer representing the number of tasks to be moved from 1
#' instance to another
#' @param exchange Exchange tasks between instances instead of moving them
#' @return A list representing the neighboring schedule
#' @export
#' @examples
#' schedule <- get.initial.schedule(3, 1:30)
#' neighbor <- move.tasks(schedule, 1)
#' neighbor <- move.tasks(schedule, 1, exchange=TRUE)
move.tasks <- function (schedule, num.tasks, exchange=FALSE) {

  # Validate args
  .validate.schedule(schedule)
  .check.if.positive.integer(num.tasks)

  # Need at least 2 instances to move/exchange tasks
  #FIXME: this check is also present in get.neighbor. Needs to be removed
  # after making this function internal so it is only called via get.neighbor()
  num.instances.in.schedule <- length(schedule)
  if (num.instances.in.schedule < 2) { return (schedule) }


  # Check if we have sufficient # tasks in the schedule (across all instances)
  if (exchange) {
    # Check if we have enough tasks to exchange
```

```
  valid <- .validate.num.tasks.in.schedule(schedule, 2*num.tasks)

  if (! valid) {
    # If not, check if we have enough tasks to move
    cat('WARN: Cannot exchange', num.tasks, ' tasks between 2 instances.
      Moving', num.tasks, 'tasks instead. \n')
    exchange <- FALSE
    valid <- .validate.num.tasks.in.schedule(schedule, num.tasks)
    if (! valid) {
      # If not, fail
      stop("Invalid argument: Insufficient number of task to move")
    } # end if - insufficient # tasks to move
  } # end if - have enough tasks to exchange?

} else {
  # Check if we have enough tasks to move
  valid <- .validate.num.tasks.in.schedule(schedule, num.tasks)
  if (! valid) {
    # If not, fail
    stop("Invalid argument: Insufficient number of task to move")
  } # end if - insufficient # tasks to move

} # end if - exchange tasks?


# number of instances to use depends on whether we are moving tasks
# or exchanging tasks
# - exchange requires 2 instances; move requires 1 instance
num.instances.to.use <- .get.num.instances(exchange)


# Get all instances with at least num.tasks tasks
all.admissable.instances <-
  .get.admissable.instances(schedule, num.tasks, num.instances.to.use)

# Can fail to get sufficient # admissable instances when:
# exchange & # instances < 2
# !exchange and # instances < 1 (due to insufficient # tasks to move in all
# instances)

if (  (exchange && (length(all.admissable.instances) < 2)) ||
      (length(all.admissable.instances) < 1) ) {
```

```
  # Insuffucient # admissable instances,
  # so try moving 1 task between instances
  cat('WARN: Insufficient # instances to move/exchange tasks.
    Moving 1 task instead. \n')
  exchange <- F
  num.instances.to.use <- .get.num.instances(exchange) # use 1 instance
  num.tasks <- 1
  all.admissable.instances <-
    .get.admissable.instances(schedule, num.tasks, num.instances.to.use)

  if(length(all.admissable.instances) < 1) {
    stop("Error: Cannot find a single instance with at least 1 task!")
  } # end if - found at least 1 instance with 1 task?

} # end if - sufficient # instances found?

idx.admissable.instances.sample <-
  sample(1:length(all.admissable.instances), num.instances.to.use)
admissable.instances.sample <-
  all.admissable.instances[idx.admissable.instances.sample]

# Remove task(s) from donor instance(s)
tasks.mat <- matrix(nrow=num.instances.to.use, ncol=num.tasks)
for (i in 1:num.instances.to.use) {

  inst <- admissable.instances.sample[i]
  num.tasks.in.instance <- length(schedule[[inst]])
  idx.tasks <- sample(1:num.tasks.in.instance, num.tasks)
  tasks <- schedule[[inst]][idx.tasks]

  schedule[[inst]] = schedule[[inst]][-idx.tasks]
  num.remaining.tasks.in.instance <- length(schedule[[inst]])
  if (num.remaining.tasks.in.instance == 0) schedule[inst] <- list(NULL)

  tasks.mat[i,] <- tasks
} # end for - loop over all instances


# TODO: need a more general way to do this
if (exchange) {
  instance1 <- admissable.instances.sample[1]
  schedule[[instance1]] <- c(schedule[[instance1]], tasks.mat[2,])
```

```
    instance2 <- admissable.instances.sample[2]
    schedule[[instance2]] <- c(schedule[[instance2]], tasks.mat[1,])

  } else {
    # Get acceptor instance
    idx.remaining.instances <-
      (1:length(schedule))[-admissable.instances.sample]
    num.remaining.instances <- length(idx.remaining.instances)
    if (num.remaining.instances == 1) { instance2 <- idx.remaining.instances }
    else { instance2 <- sample(c(idx.remaining.instances), 1) }

    # Move the task to this instance
    schedule[[instance2]] <- c(schedule[[instance2]], tasks.mat[1,])

  } # end if - move only?

  attr(schedule, 'score') <- NULL
  attr(schedule, 'runtime95pct') <- NULL
  attr(schedule, 'runtime99pct') <- NULL

  return (schedule)

} # end sub - move.tasks



#' Compare 2 schedules based on their score
#'
#' Scores are calculated for both schedules. If the score of the proposed
#' schedule is lower than the score for the current schedule, the proposed
#' schedule and score are returned. If the score of the proposed schedule is
#' greater than or equal to the current schedule, the a function of the
#' current temperature and the 2 scores is used to determine which schedule
#' to return.
#'
#' @param cur.schedule Current assigment with score attribute (list)
#' @param proposed.schedule Proposed schedule with no score (list)
#' @param runtimes Matrix of runtimes for the given instance type. Each row in
#' the matrix represents a single training sample and has 2 columns. The size
#' column is the size of task that was processed. The runtime_sec column is the
#' time taken to process the task in seconds.
```

```
#' @param runtimes.summary Numeric matrix containing mean and variance of
#' runtimes for each size
#' @param deadline Time by which job must be complete (float). Same time units
#' as runtimes
#' @param max.temp Max temperature to use in the simulated annealing process
#' '(integer)
#' @param max.iter Max # iterations to use to find the optimal schedule via
#' simulated annealing (integer)
#' @param cur.iter Value of current iteration (integer)
#' @return A list containing the accepted schedule and score
#' @export
# @examples
# data('m3xlarge.runtimes.expdist')
# setup.trainingset.runtimes('m3xlarge', m3xlarge.runtimes.expdist)
# r <- get('m3xlarge.runtimes', envir=data.env)
# rs <- get('m3xlarge.runtimes.summary', envir=data.env)
# assign('runtimes.1', r, envir='data.env')
# c.a <- get.initial.schedule(2, c(1,1,1,1))
# c.a <- get.score(c.a, r, rs, 120)
# p.a <- get.neighbor(c.a)
# a <- compare.schedules(c.a, p.a, r, rs, 120, 25, 100, 7)
compare.schedules <- function (cur.schedule, proposed.schedule, runtimes,
    runtimes.summary, deadline, max.temp, max.iter, cur.iter) {

  # Validate args
  .validate.schedule(cur.schedule)
  .validate.schedule.attributes(cur.schedule)
  .check.if.nonnegative.real(attr(cur.schedule, 'score'))
  .validate.schedule(proposed.schedule)

  .validate.runtimes(runtimes)
  .validate.runtimes.summary(runtimes.summary)

  .check.if.positive.real(deadline)
  length(deadline) == 1 || stop("Invalid argument length:
    deadline must be a single +ve real number")

  .check.if.positive.real(max.temp)
  length(max.temp) == 1 || stop("Invalid argument length:
    max.temp must be a single +ve real number")

  .check.if.positive.integer(max.iter)
```

```
.check.if.nonnegative.integer(cur.iter)
if (cur.iter >= max.iter) { stop('Invalid argument:
  cur.iter ', cur.iter, ' is >= max.iter ', max.iter) }

proposed.schedule <-
  get.score(proposed.schedule, runtimes, runtimes.summary, deadline)

cat('CURRENT.schedule: \n')
print(cur.schedule)
cat('\n')

cat('PROPOSED.schedule \n')
print(proposed.schedule)
cat('\n')

# reject all schedules that are not feasible
if (attr(proposed.schedule, 'score') < 0.95) {
  return(cur.schedule)
}

if (attr(proposed.schedule, 'processing.cost') <= attr(cur.schedule, 'processing.
  cat('PROPOSED.processing.cost <= current.processing.cost. Returning PROPOSED \n
  result <- proposed.schedule

} else {
  cat('proposed.processing.cost is higher \n')
    temp <- get.temperature(max.temp, max.iter, cur.iter)
    lhs <- round(exp((attr(proposed.schedule, 'processing.cost') -
    attr(cur.schedule, 'processing.cost'))/temp), 2)
    rhs <- round(runif (1, min=0, max=1), 2)
  cat('temp=',temp, ' lhs=',lhs, ' rhs=',rhs, '\n')

    if (lhs > rhs) {
    cat('lhs > rhs; returning PROPOSED \n\n')
    result <- proposed.schedule
    } else {
    cat('lhs <= rhs; returning CURRENT \n\n')
    result <- cur.schedule
    } # end if - lhs > rhs?

} # end if - proposed.score >= cur.score?
```

```
  return (result)

} # end function - compare.schedules



#' Get score for input schedule
#'
#' @param schedule The schedule which needs to be scored (list)
#' @param runtimes Matrix of runtimes for the given instance type.
#' Each row in the matrix represents a single training sample and has 2 columns.
#' The size column is the size of task that was processed.
#' The runtime_sec column is the time taken to process the task in seconds.
#' Used only when getting di
stribution of job runtimes by bootstrap resampling.
#' @param runtimes.summary Numeric matrix containing mean and variance of
#' runtimes for each size
#' Used only when getting distribution of job runtimes by Normal approximation
#' via Central Limit Theorem.
#' @param deadline Time by which job must complete
#' '(float; same units as runtimes)
#' @param debug Return more info when running in debug mode
#' @return The input schedule with a value for the score attribute. Score is
#' the probability of the schedule completing the job by the deadline based
#' on the training set runtimes of the tasks in the job (float).
#' @export
# @examples
# data('m3xlarge.runtimes.expdist')
# setup.trainingset.runtimes('m3xlarge', m3xlarge.runtimes.expdist)
# schedule <- get.initial.schedule(2, c(1,1,1,1))
# runtimes <- get('m3xlarge.runtimes', envir=data.env)
# runtimes.summary <- get('m3xlarge.runtimes.summary', envir=data.env)
# schedule <- get.score(schedule, runtimes, runtimes.summary, 60)
get.score <- function (schedule, runtimes, runtimes.summary, deadline, debug=FALSE)

  # Validate args
  .validate.schedule(schedule)

  .validate.runtimes(runtimes)
  .validate.runtimes.summary(runtimes.summary)
```

```
.check.if.positive.real(deadline)
length(deadline) == 1 || stop("Invalid argument length: deadline must be a
  single +ve real number")

num.instances <- length(schedule)
scores <- matrix(nrow=num.instances, ncol=3)
processing.cost <- array(dim=num.instances)

for (i in 1:num.instances) {

  tasks <- schedule[[i]]
  num.tasks <- length(tasks)

  if (num.tasks == 0) {
    scores[i] <- 1
    next;
  } # end if - any tasks on instance?

  g <- aggregate(tasks, by=list(tasks), FUN=length)

  if (num.tasks > bootstrap.threshold) {
    # cat('Using Normal approx. to runtime dist. \n')
    means <- apply(g, 1,
                   function (x) {
              runtimes.summary[which(runtimes.summary[,1] == x[1]), 2] * x[2]
          }
        )

        vars <- apply(g, 1,
                   function (x) {
            runtimes.summary[which(runtimes.summary[,1] == x[1]), 3] * x[2]
          }
        )

        job.mean <- sum(means)
        job.sd <- sqrt(sum(vars))

    # score for this instance = Prob(tasks on this instance
    # completing by deadline)
    scores[i,1] <- round(pnorm(deadline, mean=job.mean, sd=job.sd), 2)
    scores[i,2] <- round(qnorm(0.95, mean=job.mean, sd=job.sd), 2)
    scores[i,3] <- round(qnorm(0.99, mean=job.mean, sd=job.sd), 2)
```

```
      processing.cost[i] <- scores[i,2] * instance.costs[1]

    } else {
      # cat('Using boostrap approx. to runtime dist. \n')
      bootstrap.dist <-
        .bootstrap.get.job.runtime.dist(g, num.bootstrap.reps, runtimes)

      # Prob. of this instance completing by deadline
      ecdf.fn <- ecdf(bootstrap.dist)

      scores[i,1] <- round(ecdf.fn(deadline), 2)
      scores[i,2] <- round(quantile(bootstrap.dist, 0.95), 2)
      scores[i,3] <- round(quantile(bootstrap.dist, 0.99), 2)

      processing.cost[i] <- scores[i,2] * instance.costs[1]

    } # end if - more than bootstrap.threshold tasks?

  } # end for - loop over all instances in schedule

  # Return score of instance with least prob of completing by deadline
  # This determines the feasibility of the schedule
  min.idx <- which.min(scores[,1])

  attr(schedule, 'score') <- scores[min.idx, 1]
  attr(schedule, 'processing.cost') <- sum(processing.cost, na.rm=TRUE)
  attr(schedule, 'deadline') <- deadline
  attr(schedule, 'runtime95pct') <- scores[min.idx, 2]
  attr(schedule, 'runtime99pct') <- scores[min.idx, 3]
  if(debug && num.tasks > bootstrap.threshold) attr(schedule, 'norm.mean') <-
    job.mean
  if(debug && num.tasks > bootstrap.threshold) attr(schedule, 'norm.sd') <-
      job.sd
  if(debug && num.tasks <= bootstrap.threshold) attr(schedule, 'boot.dist') <-
    boot.dist

  return (schedule)

} # end function - get.score
```

```
#' Get temperature for current iteration
#'
#' @param max.temp Max value of temperature to use (float)
#' @param max.iter Max number of iterations to search for optimal solution
#' '(integer)
#' @param cur.iter Value of current iteration (integer)
#' @param method Method used to decrease temperature.
#' Currently only linear decrease of temperature with iteration is supported
#' @return Value of temperture for the current iteration (integer)
#' @export
#' @examples
#' temp <- get.temperature(25, 100, 7)
get.temperature <- function (max.temp, max.iter, cur.iter, method='linear') {

  # Validate args
  .check.if.positive.real(max.temp)
  length(max.temp) == 1 ||
    stop("Invalid argument length: max.temp must be a single +ve real number")
  .check.if.positive.integer(max.iter)
  .check.if.nonnegative.integer(cur.iter)
  cur.iter < max.iter ||
    stop('Invalid argument: cur.iter ', cur.iter, ' is >= max.iter ', max.iter)

  if (method=='linear') {
    temp <- .get.temperature.linear.decrease(max.temp, max.iter, cur.iter)
  } else {
    stop('Invalid argument: ', method,
      ' method of decreasing temperature is invalid!')
  }# end if - linear decrease in temp?

  return (temp)

} # end function - get.temperature



#' Find optimal schedule
#'
#' Want an schedule with >= .95 probability of completing job by the deadline
#' with the lowest makespan (cost)
#'
```

```
#' @param job Array of integers representing sizes of tasks in job
#' @param deadline Time (in seconds) by which job must be completed (integer)
#' @param cluster.instance.type Instance type of cluster (string).
#' All instances in the cluster are assumed to have the same instance type
#' @param cluster.size Integer representing the number of instances
#' in the cluster
#' @param max.iter Max number of iterations to use to find the optimal
#' schedule (integer)
#' @param max.temp Max temperature to use in the simulated annealing
#' process (flaot)
#' @param reset.score.pct Begin next iteration from the best schedule if the
#' difference between the best score and best score is more than this value
#' @param reset.num.iters Begin next iteration from the best schedule if the
#' number of iterations the score has not been increasing exceeds this value
#' @param debug Print debug info
#' @return A list representing the optimal schedule that could be found under
#' the given constraints
#' @export
#' @examples
#' job <- c(1,60,100)
#' deadline <- 300
#' cluster.instance.type <- 'm3xlarge'
#' cluster.size <- 2
#' max.iter <- 2
#' max.temp <- 0.5
#' data(m3xlarge.runtimes.expdist)
#' setup.trainingset.runtimes('m3xlarge', m3xlarge.runtimes.expdist)
#' best.schedule <- schedule(job, deadline, cluster.instance.type,
#' cluster.size, max.iter, max.temp)
schedule <- function (job, deadline, cluster.instance.type, cluster.size,
    max.iter, max.temp, reset.score.pct=NULL, reset.num.iters=NULL, debug=FALSE)
{

  start.time <- proc.time()

  if (!is.null(reset.score.pct)) .check.if.positive.real(reset.score.pct)
  if (!is.null(reset.num.iters)) .check.if.positive.integer(reset.num.iters)


  runtimes <- .get.trainingset.runtimes(cluster.instance.type)
  runtimes.summary <-
    .get.trainingset.runtimes(cluster.instance.type, summary=T)
```

```
cur.schedule <- get.initial.schedule(cluster.size, job)
cur.schedule <-
  get.score(cur.schedule, runtimes, runtimes.summary, deadline)

best.schedule <- cur.schedule
best.processing.cost <- attr(best.schedule, 'processing.cost')

if (debug) {
  output.prefix <- paste(cluster.size, '-inst-', length(job), '-tasks-',
    max.iter, '-SAiter-', num.bootstrap.reps, '-BSreps', sep='')
  filename <- filename <- paste(output.prefix, '.output.txt', sep='')
  sink(filename)
}

if (debug) cat('best processing.cost=', best.processing.cost, '\n')

if (debug) {
  processing.cost.timeseries <- matrix(nrow=(max.iter)+1, ncol=7)
  colnames(processing.cost.timeseries) <-
    c('Iter', paste('Acpt_', deadline, 's', sep=''), 'Acpt_95%', 'Acpt_99%',
    paste('Best_', deadline, 's', sep=''), 'Best_95%', 'Best_99%')

  processing.cost.timeseries[1,1] <- 1

  processing.cost.timeseries[1,2] <- attr(cur.schedule, 'score')
  processing.cost.timeseries[1,3] <- attr(cur.schedule, 'runtime95pct')
  processing.cost.timeseries[1,4] <- attr(cur.schedule, 'runtime99pct')

  processing.cost.timeseries[1,5] <- attr(best.schedule, 'score')
  processing.cost.timeseries[1,6] <- attr(best.schedule, 'runtime95pct')
  processing.cost.timeseries[1,7] <- attr(best.schedule, 'runtime99pct')

  filename.ts <- paste(output.prefix, '-scores-timeseries.csv', sep='')
  conn <- file(filename.ts, open='wt')
    writeLines('# Input Params', con=conn)
    writeLines(paste('# job.array = ', paste(job, collapse=';'), sep=''),
      con=conn)
    writeLines(paste('# num.jobs = ', length(job), sep=''), con=conn)
    writeLines(paste('# deadline = ', deadline, sep=''), con=conn)
    writeLines(paste('# cluster.instance.type = ',
      cluster.instance.type, sep=''), con=conn)
```

```
    writeLines(paste('# cluster.size = ', cluster.size, sep=''), con=conn)
    writeLines(paste('# max.iter = ', max.iter, sep=''), con=conn)
    writeLines(paste('# max.temp = ', max.temp, sep=''), con=conn)
    writeLines(paste('# reset.score.pct = ', ifelse(is.null(reset.score.pct),
      'NULL', reset.score.pct), sep=''), con=conn)
    writeLines(paste('# reset.num.iters = ', ifelse(is.null(reset.num.iters),
      'NULL', reset.num.iters), sep=''), con=conn)
    writeLines(paste('# debug = ', debug, sep=''), con=conn)

    write.table(t(processing.cost.timeseries[1,]), file=conn, sep=',', quote=FALS
      row.names=FALSE)
    flush(conn)
} # end if - debug?


# go from 0 to 1 less than max.iter
# so we start at max temp and end just above 0 and avoid divide-by-zero errors
for (i in 0:(max.iter-1)) {

  if (debug) cat('\n\n==========\nSA iter: ', i, ' (', (i+2), ') \n',
    '==========\n\n', sep='')

    proposed.schedule <- get.neighbor(cur.schedule)
    cur.schedule <- compare.schedules(cur.schedule, proposed.schedule,
      runtimes, runtimes.summary, deadline, max.temp, max.iter, i)
  cur.processing.cost <- attr(cur.schedule, 'processing.cost')

  # update best score, if necessary
  if (cur.processing.cost < best.processing.cost) {
    best.schedule <- cur.schedule
    best.processing.cost <- attr(best.schedule, 'processing.cost')
  } # end if - cur schedule better than best schedule so far?

  if (debug) cat('best processing.cost=', best.processing.cost, '\n')

  # restart from current best schedule if score of current schedule
  # is too low
  if (!is.null(reset.score.pct)) {
    d <- (cur.processing.cost - best.processing.cost)
    d.pct <- 100*d/best.processing.cost
    if (d.pct > reset.score.pct) {
      cur.schedule <- best.schedule
```

```
      if (debug) cat('Resetting current schedule to best schedule since
        d.pct=', d.pct, '. Best processing.cost so far = ', best.processing.cost,
    } # end if - reset current schedule to best schedule
  } # end if - reset.score.pct defined?



  if (debug) {
    processing.cost.timeseries[(i+2),1] <- (i+2)

    processing.cost.timeseries[(i+2),2] <- attr(cur.schedule, 'processing.cost')
    processing.cost.timeseries[(i+2),3] <- attr(cur.schedule, 'runtime95pct')
    processing.cost.timeseries[(i+2),4] <- attr(cur.schedule, 'runtime99pct')

    processing.cost.timeseries[(i+2),5] <- attr(best.schedule, 'processing.cost')
    processing.cost.timeseries[(i+2),6] <- attr(best.schedule, 'runtime95pct')
    processing.cost.timeseries[(i+2),7] <- attr(best.schedule, 'runtime99pct')

    write.table(t(processing.cost.timeseries[(i+2),]), file=conn, sep=',', quote=
      row.names=FALSE, col.names=FALSE, append=TRUE)
    flush(conn)
  } # end if - debug?

} # end for - loop over all iterations


# sort task.sizes in each instance
# for (i in 1:length(best.schedule)) {
#   best.schedule[[i]] <- sort(best.schedule[[i]], decreasing=TRUE, na.last=NA)
# } # end for - loop over all instance

if (debug) attr(best.schedule, 'scores.ts') <- processing.cost.timeseries

cat('\nBest processing.cost: ', attr(best.schedule, 'processing.cost'), '\n')
cat('Best schedule: \n')
print(best.schedule)
cat('\n\n')

d <- proc.time()-start.time
cat('Time taken: ', d[3], ' seconds')

if (debug)  {
```

```
    sink()
    close(conn)
  } # end if - debug?

  return (best.schedule)

} # end function - schedule
```